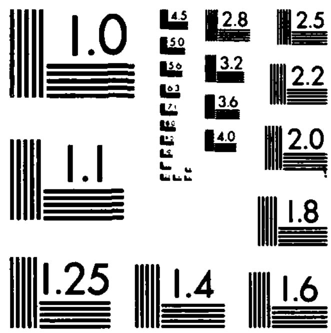
AD-A184 005   ADAMEASURE: AN IMPLEMENTATION OF THE HALSTEAD AND HENRY METRICS(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA P M HERZIG JUN 87   1/2

UNCLASSIFIED   F/G 12/5   NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

ADAMEASURE:
AN IMPLEMENTATION
OF THE HALSTEAD AND HENRY METRICS

by

Paul M. Herzig

June 1987

Thesis Advisor: Daniel L. Davis

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION: Unclassified	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; Distribution is unlimited
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION: Naval Postgraduate School; 6b OFFICE SYMBOL (If applicable): Code 52	7a NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
6c ADDRESS (City, State, and ZIP Code): Monterey, California 93943-5000	7b ADDRESS (City, State, and ZIP Code): Monterey, California 93943-5000
8a NAME OF FUNDING/SPONSORING ORGANIZATION; 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (City, State, and ZIP Code)	10 SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

11 TITLE (Include Security Classification)
ADAMEASURE: AN IMPLEMENTATION OF THE HALSTEAD AND HENRY METRICS (u)

12 PERSONAL AUTHOR(S)
Herzig, Paul M.

13a TYPE OF REPORT	13b TIME COVERED	14 DATE OF REPORT (Year Month Day)	15 PAGE COUNT
Master's Thesis	FROM ____ TO ____	1987 June	184

16 SUPPLEMENTARY NOTATION

17 COSATI CODES			18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Ada Metric; Data Flow Complexity; Ada Measure; Halstead; Henry

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

A software metric is a tool that should be used in the development of quality software. The properties that define good software vary but encompass reliability, complexity, efficiency, testability, understandability, and modifiability. The Henry metric measures the complexity of data flow within a module and the complexity of inter-module communication. This thesis is an extension of a previous thesis titled 'AdaMeasure' that calculated the Halstead metric. The present design and implementation is a tool that computes the Halstead and Henry metrics for Ada programs.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT: ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION: Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL: Prof. Daniel L. Davis	22b TELEPHONE (Include Area Code): (408) 646-3091; 22c OFFICE SYMBOL: Code 52Dv

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Approved for public release; distribution is unlimited.

AdaMeasure
An Ada Software Metric
Implementation of the Henry Metric

by

Paul M. Herzig Jr.
Lieutenant Commander, United States Navy
B.S.E.E., University of New Mexico, 1976

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

June 1987

Author: P.M. Herzig

Approved by: Daniel L. Davis, Thesis Advisor

Bruce J. MacLennan, Second Reader

Vincent Y. Lum, Chairman, Department of
Department of Computer Science

Kneale T. Marshall
Dean of Information and Policy Sciences

ABSTRACT

A software metric is a tool that should be used in the development of quality software. The properties that define good software vary but encompass reliability, complexity, efficiency, testability, understandability, and modifiability. The Henry metric measures the complexity of data flow within a module and the complexity of inter-module communication. This thesis is an extension of a previous thesis titled 'AdaMeasure' that calculated the Halstead metric. The present design and implementation is a tool that computes the Halstead and Henry metrics for Ada programs.

TABLE OF CONTENTS

I. INTRODUCTION AND BACKGROUND

A. DEFINITIONS

A metric is an assignment of indices of merit to programs in order to evaluate and predict software quality [Ref.1: p.6-2]. The qualities to measure are, at present, subjectively chosen but in general encompass reliability, complexity, efficiency, testability, understandability and modifiablity [Ref.2: p.1-3]. The predictive nature of a metric allows it to be used to say "when" to proceed to the next phase in the software life cycle model. Another aspect of the predictive nature of a metric would be for it to provide management with a rough guess of the outcome of a particular path of development, provide an acceptance index, or provide an immediate feedback loop to the implementors while in the unit test phase [Ref.2: p.5]. How the metric is implemented will dictate its primary use from the above selections.

B. SALLIE HENRY'S METRIC

Sallie Henry's metric attempts to measures data flow complexity. It is intended to be used as a tool to establish a module's quality or to enforce particular modularization standards [Ref.2: p.6]. She argues that quality control of software is the result of software reliability and that reliability comes about through well designed modules that do not have complex data flow.

The hierarchical structure of a program should be layered modules. Each layer should function as a virtual machine and be composed of modules. This approach to modularization gives each module characteristics that can be exploited so that each module can be independently developed, more easily comprehended, assembled so that the system is more stable and designed so that the system is a great deal more flexible. This schema of development extols two primary tenets that are stated by D. Parnas in [Ref.3: p.339] and quoted here:

> . . . provide the user of a module with all the information to use the module correctly, and nothing more. Provide the implementor of the module with all the information to implement the module correctly, and nothing more.

All this implies that a good design will have high module cohesion, good module strength and low module coupling [Ref.4: p.330].

C. INFORMATION FLOW

Information flow complexity is a twofold process the flow of data within a module and the flow of data external to the module. The measurement of these criteria is dependent on two premises: (1) that there is a capability to measure this data and (2) the data obtained can be used to evaluate software design. The seemingly obvious nature of the first premise runs into problems in implementation and applicability, but if it is accepted that the first deficiency can be surmounted, then the second part remains to be shown as reasonable. Applicability is a debated concept that is still not resolved. It revolves around whether the data gathered is related to the property under consideration. It is

further exacerbated by the human element that defines an environmental bubble and then programs within this bubble. How to measure this bubble without destroying its foundations is the problem of measuring human performance. The problem of what to measure is the problem of applicability.

The more specific the metric's application the less the applicability property is questioned, but, the problem of "what" to measure is still not clearly defined. This thesis will not argue the applicability question because the approach of Sallie Henry is reasonable and the results obtained from the metric appear to adequately encompass the area of data flow complexity. If the reader will accept that the properties measured are related to data flow complexity then the results obtained are also related to complexity.

The second premise is even more thorny. If the data is obtained and it seems reasonable can it be shown to be truly the result of the property under measurement? Any human endeavor will never be clearly and objectively quantified. Thus, the answer to the efficacy of the second premise is, proceed and maybe the amassing of results will eventually show the correlation.

The above analysis is far from a convincing argument to utilize metrics to measure programs however as this thesis was developed the applicability of measuring data flow complexity in order to determine code quality became more apparent although not proven. Nothing will be learned if no attempt is made to measure data flow complexity. This thesis attempts to measure data flow

complexity in the light of learning and the hope that the data gathered will prove the applicability of the process.

Consider first a simple module: a procedure in a structured language. Each procedure defines certain relations between itself and other procedures. These include:

- formal input/output parameters
- function call input and return data
- local data structures
- global data structures

These relations will generate a particular information flow structure similar to a hierarchical tree structure. This tree structure is peculiar to the procedure and will reflect its complexity of structure. It is reasonable to analyze this tree to determine derived calls, local data flow and global data flow.

D. RELATIONS

Some definitions are now in order. Global data flow exists from procedure 1 to procedure 2 if procedure 1 deposits data in the global data structure and then procedure 2 reads that data. Local data flow comprises direct and indirect species. A local direct flow, from procedure 1 to procedure 2, results when procedure 1 calls 2 passing parameters. An indirect data exchange from procedure 1 to procedure 2 exists if procedure 2 calls 1, which returns a value used by 2, or procedure 3 calls both 1 and 2, and passes an output value from 1 to 2.

Figure 1 represents data flow from procedure to procedure or from a procedure into a data structure. Parameter passing within this scheme is represented by the arrows. A hidden data exchange through modification of a variable is represented by the dashed flow arrow. Module A retrieves data from the data structure then calls B passing a parameter; module B updates the data structure. C calls D passing a parameter. D calls E with a parameter and E returns a value to D which is used by D and passed to F. The function of F updates the data structure.

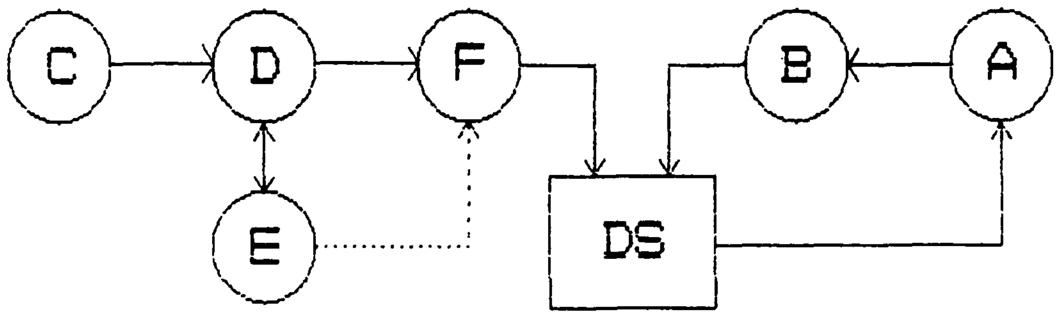


Figure 1. Data Flow Structure

The direct data flows represented are:
A -> B, C -> D, D -> E, D -> F.

These are simply the calls.

The indirect local flows are:
E -> D, F -> A.

The global flows are:
B -> A, F -> A.

Both B and F update the data structure while A retrieves data from the structure.

The implications of data flow for procedure and function calls will be discussed later with derived calls.

The calling notation A(x) -> B() or A() -> B(x) is used to connote a data flow transmission from A to B either by direct parameter passing or side-effect. In the first condition the variable x is returned to procedure A and in the second example the variable x is sent to B. A condition that leads the Henry metric to not detect a procedure or function call's data flow (labeled a missed call) is for the condition where A(x) -> B() and variable x is a returned value from B not modified within procedure A's code. An example of this would be a conditional statement within A that depends on the returned value from function B. The data flow detection problem leads to two key ideas, effective parameters and data utilization.

Calls that are detected by information flow analysis are dependent upon how the information is passed. If the conditions A() -> B(x) exists where parameter x

is passed to B or condition A() -> B() where no parameters are exchanged then the calls will not be missed if B receives information in one of the following formats:

- a formal parameter
- a data structure
- a constant
- an actual parameter from a third procedure whose value is modified within A prior to the call to B

An effective parameter will define the call structure in such a way that the data flow will not be missed. It is a parameter that receives information from one of the calling procedure's parameters, a data structure, a constant, or a third procedure's returned actual parameter that is modified within the calling modules structure. What the effective parameter implies is that side- effect data flow is difficult to effectively analyze. Another construct that will cause a missed call is the condition A(z) -> B(x) where B is a function. This condition means A uses data from B. A uses data from B if (1) B updates a data structure used by A; (2) A receives a constant from B; (3) A receives an output parameter from B; or (4) B updates a return value to A. Thus information flow will be detected if A passes B an effective parameter or if A uses data from B.

Appendix A gives all the rules that are applicable to the data flow relationships. Some notation is now needed to simplify the descriptions that follow.

The form of a relation is L <- R1, R2, R3,...Rn; where L is the resultant from the application of the relationships R1, R2, ... Rn. An example would be:

A.D3 <- A.D1, A.D2, A.constant.

This series notation represents the code line that begins with D3 below.

```
A()
 begin
  .
  .
  D3 := D1 + D2 + 1;
 end procedure A;
```

In words, the A.D3 means procedure A updates data structure D3 by first applying relationship A.D1 then A.D2 and finally A.constant. This format shows that data flows into procedure A's data structure D3 from the noted relationships. A thorough discussion of the notation for the relations is given in Appendix A but a short discussion follows to aide in the immediate understanding of Figure 2.

The notation B.1.I defines the first input parameter in the actual parameter list of procedure B and an O would refer to an output parameter. All possible data flow paths are considered even if a B.1.I parameter is not an input parameter. Thus, if procedure B has an output actual parameter in position B.1 and the Henry metric attempts to analyze this parameter as an input flow an error condition would result from the attempted evaluation (depicted as B.ERROR). B.NULL means that no relationship exits for this parameter or that there is no data flow into or out of the parameter being considered.

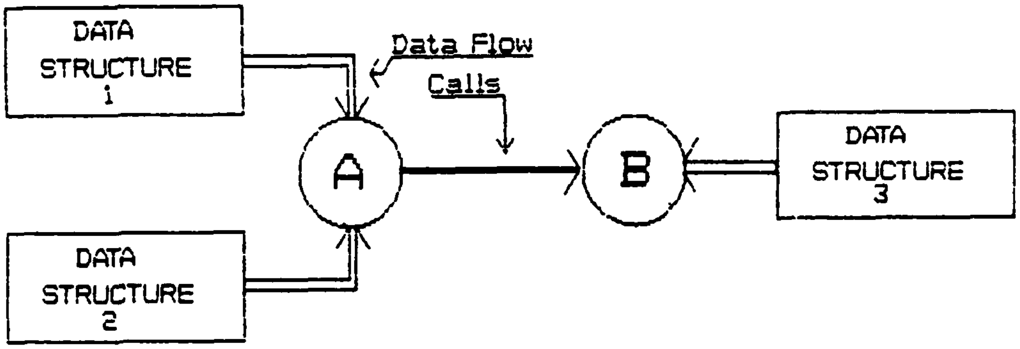


Figure 2. Data Flow With Call Structure

Code

```
A()
 begin
  X := D1 + 1;
  Y := D2;
  B(X,Y);
 end;


B(P,Q)
 begin
 D3 := P + Q;
 end;
```

Relations Set

A1 B.1.I <- A.D1, A.CONSTANT
A2 B.2.I <- A.D2
B1 B.1.O <- B.NULL
B2 B.2.0 <- B.NULL
B3 B.D3 <- B.1.I, B.2.I

The relation sets were derived by looking at the data flow into and out of procedure A. That is, since procedure A has no parameters there can only be local data flows into or out of the procedure. These flows are described in terms of the procedure call to B. B.1.I stands for procedure B's first input parameter. This parameter is fed from procedure A's data structure D1 and a constant. Analyzing procedure B's second input parameter yields the A2 relationship. Relationship B1 describes the first parameter in procedure B as an output parameter to procedure A that receives no data for transfer. Relationship B3 describes how the two input parameters to procedure B constitute the data flow to this data structure.

The data flow analysis deals primarily with the analysis of parameters which are direct data flow and indirect data flow as defined above. Modifying Figure 2 and incorporating some local variables will illustrate some more data flow analysis techniques as seen in Figure 3.

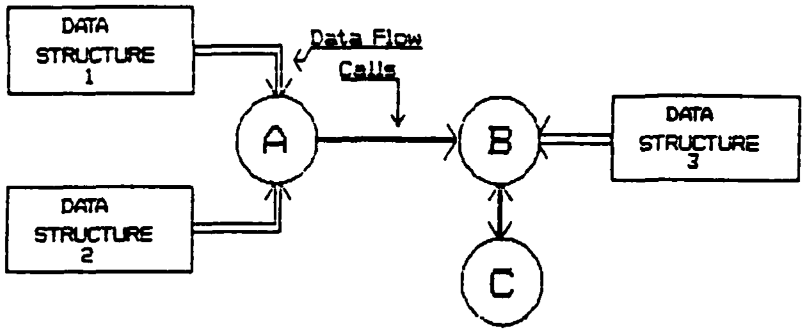


Figure 3. Data Flow And Inter-dependent Procedures

Code

```
A()
 begin
  X := D1 + 1;
  Y := D2;
  B(X,Y)
 end;


B(P,Q)
 begin
 R := Q;
 C(P,R,S);
 D3 := S;
end;


C(I,J,K)
 begin
  K := I + J;
  J := J + 1;
 end;
```

Relation Set

A1, A2 same.

```
B1  B.1.O <- C.1.O
B2  B.2.O <- B.NULL
B3  C.1.I <- B.1.I
B4  C.2.I <- B.2.I
B5  C.3.I <- B.ERROR
B6  B.D3  <- C.3.O
```

```
C1  C.1.I <- C.NULL
C2  C.2.O <- C.2.I, C.CONSTANT
C3  C.3.O <- C.1.I, C.2.I
```

In the relation set B1 receives data from procedure C's output parameter. B2 is the same. B3 through B5 describe the parameter list of procedure C. However B5 denotes an error or a condition that is not allowed. That is, the data direction was in error as variable S is an output from procedure C as indicated by relation B6. It should be noted that this relation set building considers all possible data flow paths without regard to the possibility that the parameters could be assigned only particular directions as Ada formal parameters are. Figure 4 shows the effects of a function call.

Code

```
A()
 begin
  X := D1 + 1;
  Y := F(X);
  B(X,Y)
 end;
```

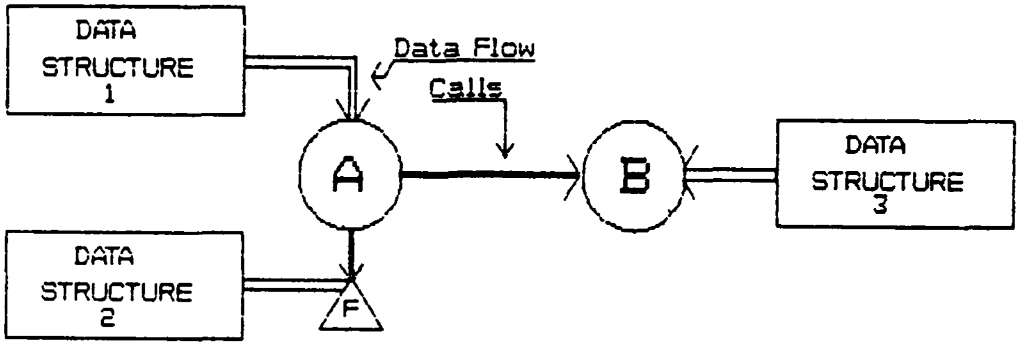


Figure 4. Data Flow With Function Call

```
F(M) return integer;
 begin
 N := D2 * M;
 return N;
end;
```

Relation Sets are changed as follows:

A1 F.1.I <- A.D1, A.CONSTANT
A2 B.1.I <- A.D1, A.CONSTANT, F.1.O
A3 B.2.I <- F.O

F1 F.1.O <- F.NULL
F2 F.O <- F.D2, F.1.I

Relation A1 has changed to reflect the analysis of the function call to F. The input to the function call is analyzed as well as its output and any possible modification of its input parameter. This analysis can be seen to cover all possibilities of hidden data transfers except the missed calls described earlier.

E. INFORMATION FLOW STRUCTURE

Once the relation set has been built the relations are sorted alphabetically and stored for future use in the Information Flow Structure (IFS). A recursive algorithm is employed to build the information tree structure for the flow analysis. The IFS is then analyzed to find the derived calls, the local flows and, finally, the global flows.

The IFS will have leaves that are data structures; the root is the initial call from the highest level procedure. Each node of the tree will have the relational form of X.DS, X.O, X.k.I, or X.k.O. See Appendix A for all the possibilities of derived calls. The local flows are described in Appendix A as derived calls. The global flows for a particular data structure are all the possible paths from leaf elements of the form A.DS to the root.

F. INDICES OF MERIT

The calculations of the indices of merit use the idea that the complexity of a module comprises the complexity of the code plus the complexity of the connections of the code to other modules. The formula describing the complexity of a module is

Complexity = length * (fan-in * fan-out) ** code index.

Length is defined as the number of executable statements. The expression fan-in * fan-out represents all the combinatorial possibilities for each input to produce an output. The code index is an exponent that represents the code

difficulty. Nominal code difficulty for operating systems is 2. This index needs more data for other types of programming.

The purpose of this computation is to produce comparative numbers of merit that point out and isolate specific areas within the code that have the potential for problems. A high fan-in/fan-out implies a large interconnection to outside modules. This leads to the assessment that the code in question is most likely not properly modularized or, more succinctly, that the code has more than one function. The other form of data flow is global data flow to data structures. It is calculated as follows:

$$\text{Global flow} = \text{write} \;\; * \; (\text{read} + \text{read\_write}) + \text{read\_write} * (\text{read} + \text{read\_write} - 1)$$

The term write refers to a change to the data within the structure through an assignment statement and a read is an access to the data structure that does not change the data. The identifier read-write is the sum of reads and writes. A high global flow implies overworked data structures and represents a stress point in the program. A stress point is the weak link in the chain. The presence of high flow is not automatically an indicator of poor programming but it is a juncture in the program that is highly susceptible to problems. Once the metric has assembled all the different components. such as fan-in or global reads and calculated the above equations it performs module analysis.

G. MODULE ANALYSIS

Module analysis revolves on the outputs of each of the above equations and their respective components. The numbers generated are symptomatic of certain problems. The analysis is first conducted with the equations output defining the particular categories of problems then the components refine the analysis. Examples of the first level of analysis follow:

A high global flow calculation implies an overworked data structure. These structures are overworked because of the need for continuous accessing. This implies a better decentralized design is in order, that is, distribute the information to the procedures that it serves. A high module complexity index indicates not enough modularization. This number is to be treated with respect but should be analyzed in context with global data flow. Together these indices represent the in's and out's of the modules data. A corrective action based solely on complexity should be avoided. A procedure should be analyzed for singularity of purpose and non-duplication within a module. Simply put, a procedure should be in one place, have one purpose and have minimal external references. These properties are quantified by the Complexity and Global flow metric numbers.

Next the interim cases where one aspect is high and the other component is low. A module with high global flow and low complexity shows poor internal structure. This structure will most likely have excessive numbers of procedures with extensive use of data structures outside the module. Low global flow with

high module complexity implies either poor decomposition into procedures or extremely complicated interface.

H. INTERFACE MEASUREMENTS

Interface between procedures comprise protocol interface, coupling and binding of procedures. Protocol interface from module A to B is defined as those procedures that are not in any other module and which receive information from A for passing to B. Binding is the sensitivity measure between modules, that is, tightly bound modules have a high sensitivity. A tightly bound module is difficult to change without adversely affecting the other module. Coupling is the strength of binding. Figure 5 depicts the interface structure.

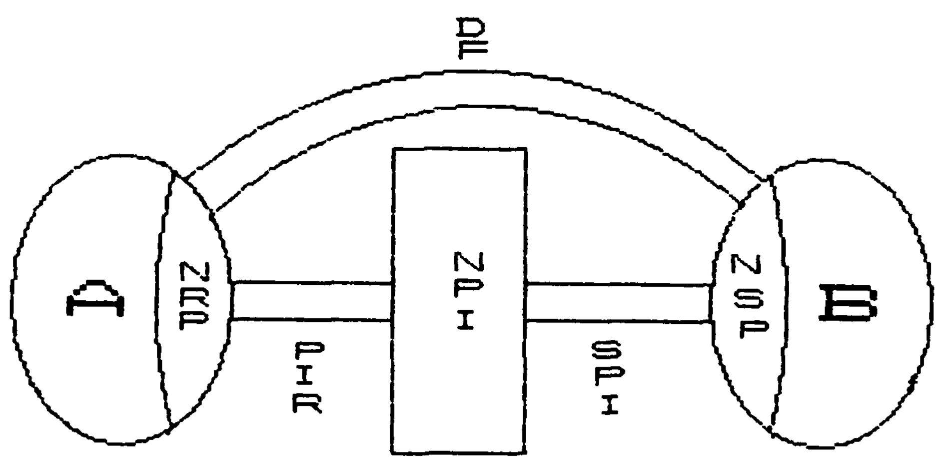


Figure 5. Interface Structure

Protocol interface, since it is not symmetrical between procedures, requires the construction of a tabular cross reference table with all possible procedures on both the X and Y axes. The internals of the table are the data flow complexity indexes for one way transmission from each procedure to the other.

Figure 5 shows that binding is sectioned into five components; the number of procedures sending information from A (NSP), the number of procedures receiving information from B (NRP), the number of procedures in the protocol interface (NPI), the number of paths to the interface from A (SPI), and the number of paths from the interface to B (PIR). The Direct Flow paths represented as the outer loops are data transmissions without the interim procedures. [Ref.2: p.85] lists the binding calculations as follows:

Binding = (NSP + NPI) * SPI + (NPI + NRP) * PIR

The term (NSP + NPI) * SPI is the coupling strength.

All the direct path binding is calculated by

DF Binding = (NSP + NRP) * DF

Modules that are tightly bound are extremely difficult to maintain and modify. This difficulty stems from their lack of independence and the "ripple effect" of changes to one module flowing into the other.

I. THEORY SUMMARY

The purpose of the Henry metric is to provide designers and implementors with a method to quantify the quality of the code that they are developing. The goal is to produce reliable code that is interconnected in as logical a fashion as possible. Information flow complexity produces reliability through enforcement of design rules that lead to well connected code. The measurements will point out lack of functionality, improper modularization, poorly designed modules, poorly designed data structures, system stress points, inadequate refinement, strength of binding, modifiability, missing levels of abstraction and, will produce comparative indices to assess changes.

II. DESIGN CRITERIA

A. INTRODUCTION

The design of the implementation of the Henry metric was a two step process. First, a thorough understanding of the previous work by Neider and Fairbanks [Ref.5: pp.1-164] was undertaken to determine what data were available for importation into the Henry metric. The intention of the study and the basic design issues are (1) modify the output of the parser portion of their thesis to include the necessary data passes to the Henry metric (2) to initially analyze only the Ada package as a unit (3) encapsulate as much of the Henry metric into one Ada package as possible and (4) calculate the necessary Henry metric numbers transparently to the user but present the user with an output that is easily understood.

The underlying premise, of this program, is that the code presented for analysis has been successfully compiled. If the code does not compile and is presented to the parsers it will most likely fail to parse but in the event it does escape detection it will be erroneously analyzed.

The design criteria of encapsulation of the Henry metric was modified during the implementation due to the unwieldy length of the code. The division yielded three packages: one that holds all the global data, another for the analysis portion and a third for the interface to the user. Although this division violated one of

the basic design issues it was necessary in order to achieve solid data transfers between the Naval Post Graduate School computer and the Naval Weapons Station computers.

While the level of what was available from the parsers was being determined the data structures of the Henry metric were being layed out. The data structures and data gathering procedures for the Henry metric were designed to be as simple, yet as flexible as possible. Once the layout of the linked list data structure to hold the raw information for the Henry metric was decided upon, the tedious procedure of inserting the appropriate calls to the Henry package was undertaken. Basically, this reduced to exercising all the possible data flow characteristics from function or procedure calls that the Henry metric could expect to encounter and then ensuring that an appropriate call to the Henry data collection procedure was placed in the Neider/Fairbanks parser.

Next, the analysis procedure was designed. The analysis was separated from the display because computers have very different capabilities in their output devices. The first package analyzes the collected raw data and the second displays the finished, smooth data.

The program is menu driven. It initially gives the user a choice to parse a new program or view old data. If the parse choice is selected the parsers feed both the Henry and the Halstead packages with data. After a successful parse the user is presented with a choice of either viewing the Halstead or Henry metric data. This is when the analysis portion of the Henry metric is called. It is not

until the user decides which data to view is there a distinction made as to how to process the parsed data. This feature was designed for several reasons:

- Use the Halstead metric to refine the code
- then analyze the code via the Henry metric for data flow
- but the user should still have the option to select either metric

The data for both metrics are essentially different as are the purposes for gathering the data but the final goal of this dual metric system is to produce good code.

Finally, the presentation data module was designed. The Halstead and the Henry metric both produce numbers which are essentially meaningless unless a thorough understanding of the particular metric implementation is undertaken. Thus, both metrics, in differing fashions present "help" data to aid understanding of the metric output. These files are both verbally and graphically presented to the user.

The overriding design issue was to modularize the Henry metric as much as possible. The most significant exception to the Parnas' ideal [Ref.4: p.330] are the numerous calls to the data gathering procedure from the parser modules. These calls depend on details of how the data will be analyzed in their sequencing and data passing scheme. A conscientious effort was made to minimize global data and isolate procedures into nearly stand alone modules.

B. SPECIFIC ISSUES

The housekeeping routines of the global package are used to adjust the parsed data into a more palatable form for the analysis and presentation procedures. The output from the parser is stored as a linked list of raw data produced by procedures WRITE_HENRY_DATA and CREATE_NODE. This data is then analyzed by the ANALYSIS PACKAGE for the particular data constructs that represent a data flow. The output is an array of tabulated data that is a set of all the relations necessary for the Henry metric to detect local and global data flow.

The display module presents data in a tabular format or as a graphical representation for relative merit analysis. The intent was for the user to see the effect of changes, or to select a more verbose description of the meaning of the results. The modules that accomplish tabular and graphic displays are separated again because of the varying capabilities of machines. The purpose of these procedures is to provide some form of relative measure to the user so that the improvement or results of a change could be more objectively weighed. The overall purpose of the display modules is to show the data in such a fashion that an intelligent assessment is possible.

C. DESIGN ISSUES

Design issues encountered in the implementation of Henry's metric involve the efficient use of the Ada language's structures and data analysis techniques. Sallie Henry developed a metric process in which the constructs of a particular

language are ignored or not put to specific use. That is, some languages have extremely thorough type and range checking facilities. This is not considered in the basic design of her metric. These powerful features are incorporated in Ada and provide the application programmer more analysis capability.

This design issue concerns the collection of 'all possible paths' data for analysis of actual parameters. The approach taken by Henry is biased to a language where input, output, and combination input output parameters are treated as if they could be modified by the particular procedure regardless of their type. Ada is very picky about the manipulation of formal and actual parameters and goes to great lengths to ensure that parameter consistency is maintained by means of strong type/range checking. The explicit declaration of a parameter type was used to select which component of the complexity equation should be updated. The appropriate fan-in or fan-out number was also correctly updated from default declarations such as the undeclared default formal parameter.

The data analysis technique issue encountered was the need to analyze the data via the IFS. Henry's IFS was designed so that a traversal of its nodes analyzing parent-child pairs will capture all the transitive relational data flows. The transitive flow analysis designed into the present parser will account for the first two layers. The reasoning behind this approach stems from a program review. This review, albeit not extensive, was conducted looking for the predominant use of transitive relations. The review revealed that transitivity is not often used and if used is at most two layers deep. There was little use of deep

transitive constructs. Thus, the design approach selected will detect the majority of the transitive data flow paths without the need for an extensive tree structure. The "normal" program has few transitive relations but the capability to analyze this style of program would add more accuracy to the metric.

Another design anomally of the metric is the problem of detecting the difference between a function call outside the package declaration and a global data structure manipulation. Ada libraries or packages inhibits the proper analysis of a function call as opposed to a data structure read unless a full compiler's output is available. The present metric was designed so that local function calls (within the package being analyzed) are properly valued but the function calls outside the package are treated as data structure manipulations or more specifically as global data flows.

D. CONCLUSION

The design and implementation phase was driven by the analysis of the Neider/Fairbanks parser portion of their thesis followed by the modularization of the Henry metric. The tradeoffs considered were: the strong typing and range checking of the Ada language, the need for an information flow tree, the need for relative output for the user and, and most importantly the desire to incorporate all of the Henry metric into one Ada package.

III. DESIGN AND TESTING

A. THE EMBEDDED CODE

The previous work done by Neider/Fairbanks had to be modified to output the necessary data for the Henry metric. This was accomplished through embedding calls to the Write_Henry_Data procedure in Parser0, Parser1, Parser2, Parser3 and Bypass Function (See Appendix C). The writing of the Lexeme, or identifier's name, was controlled by a Boolean that was turned on or off according to the position of the parse of a particular package. The design criteria was to keep the data gathering as simple as possible. If time permitted, a more thorough and sophisticated scheme could be developed. The embedded code was thoroughly tested by two test harnesses that simulated a series of Current_Token_Records in the form of an input Ada package.

B. THE HENRY PACKAGE

The first package to be implemented was Henry.pkg. It was conceived to be a stand-alone construction that would initialize the data collection process, receive data from the other parsing packages and store the raw incoming data in a linked list. (See Appendix B). Minimal variables and foreign procedures from other packages are used. The Henry package's only "withed" packages are TEXT_IO, HENRY_GLOBAL, HENRY_ANALYSIS and HENRY_DISPLAY. This

approach was considered necessary so that the subsequent changes or upgrades would not affect other modules (ripple effect). The design was to implement a basic Henry metric first for Ada packages then to improve and more fully develop the Henry analysis techniques if time allowed. The Main Menu module sequences the user into the analysis and display support packages. The modularization was considered necessary because the analysis and display packages are separate entities and the separation will ensure maintainability.

The initialization is conducted by procedure Initialize_Henry and the declaration statements that assign initial values to various Boolean variables. Initialize_Henry creates two head nodes, one for the raw data linked list, the other for the procedure or function length records. The raw data linked list storage is a straight line of Henry records. These records have five fields that identify whether this is (1) local or global declaration, (2) the variable/procedure's name, (3) an action class, (4) a parameter class and (5) a pointer to the next record. The action class is comprised of various identifiers that range from procedure type to end parameters declare. Their purpose is to delineate the actions within the parsed program so that the Henry analysis package can look for the data flow. The parameter type field is used to define input, output or combination input output formal parameters. The variable Henry_Line_count is purposely initialized within this procedure to draw attention to it's initial value. The array of length records is initially a parallel construct not directly tied to the procedure or function it holds the data for. In the analysis package a sequential process is

produced where the records are linked to the data manipulation array. The purpose of the length record is to hold the begin and end line counts of each procedure or function. These line counts are used later to compute the specific modules length for inclusion in the complexity equation.

The receipt of incoming data is accomplished primarily by Write_Henry_Data. This procedure is supported by a boolean Write_Henry_Enable. This boolean turns on or off the recording of the incoming records from the Get_Current_Token_Record procedure. Specifically, the boolean will allow recording only selected data from the incoming record stream selected by the place within the recursive descent parser that the boolean is activated. This control is necessary to pick and choose the data that is critical and to ignore the remainder.

The procedures Create Node and Clear Henry Lexeme support the data gathering scheme. The "in out" pointers within Create_Node serve the purpose of allowing a view of the last record in the incoming stream or to work on the current record. It is arranged so that New Node points to the newly created blank record and Last Record points to the just filled in trailing record. Procedure Clear Henry Lexeme is necessary because of the way Ada handles strings. Create Line Node procedure functions identically to Create Node.

The incoming data is chosen from within the Bypass Function and from Parser0 to Parser4 [Ref.5: pp.102-160] by where the calls to the Write henry data procedure is positioned. The purpose of this approach is to assure the Henry

metric receives sufficient information but more importantly that the records writen into the linked list are delimited in a particular fashion for ease of analysis. There is still considerable data that can be collected for analysis from the parsers but the Henry metric is not to the stage of development where it would be useful. The added depth of information could be used in two areas: analysis and a more informative output from both metrics.

The Write Henry Data procedure selectively enters the field data into the raw data linked list records as dictated by the incoming actual parameters. That is, the incoming data has default settings but if the data is to be ignored then the "null setting" is passed as an actual parameter. This assists in the gathering process. The design of the data gathering modules is such that modifications could be easily implemented. This was purposely designed into them so that upgrades would be fairly painless.

The Henry.pkg was constructed with modularization and maintainability in mind. It was meant to be a stand alone entity that receives data from the Neider/Fairbanks Bypass Function and Parser packages. It performs the functions of initialization, data receipt and data storage besides defining the data structures used throughout the Henry metric packages. There are a number of improvements that could be added to the actual parameter analysis. These improvements all concern the wealth of options Ada provides in parameter passing schemes, such as, aggregates, dot notation to access hidden variables and allocators. Further, the present Henry metric does not analyze the incoming

actual parameters for expressions but the variables are all considered for inclusion in the complexity calculation by the transitivity analysis.

C. HENRY ANALYSIS PACKAGE

The Henry Analysis package comprises three procedures to set up the raw linked list data and a fourth procedure to actually analyze the code for metric calculations. The Analysis procedures are called sequentially from the Henry package and function as support for the Henry package. They operate on the data in sequential discrete steps. They first determine the formal parameters, then search and identify procedures and variables and then determine the metric numbers. The approach used was to nibble each piece of the tremendously complex data flow calculation down into minute sub-steps until all that is left is to simply count the marks on each record for determination of the complexity or global flow metric numbers. This approach removed the necessity for an arduous single pass calculation.

The set up procedures are CLEAN_UP_HENRY_DATA, SET_UP_HENRY_ARRAY, and SPRUCE_UP_HENRY_DATA. A support function, LOCAL_NAME, assists in the setting-up process. These procedures' end product are two metrics, the complexity metric and the global flow metric.

The Clean Up procedure ensures that all parameter type records have all their fields properly filled. It scans for their parameter lists all the procedures and functions that are declared in the analyzed package . The field of most

importance is the classification of either "in, out or in-out" type parameters. These fields are checked up to the colon delimiter within the formal parameter list and then entered into all parameter_type records correctly.

The Set_Up procedure scans through the entire linked list setting up another array of pointers to facilitate the analysis process. The Henry_Array records have identifier, beginning pointer and line_length record pointer entries. This procedure's purpose is to break up the long linked list into another array. It actually does not sub-divide the list it merely arranges an array of pointers into the linked list that delineate each function or procedure. The resulting array is called the Henry_Array. The line length record pointers are records that hold the stop and stop line numbers. These records are eventually used to compute procedure/function lengths.

The Spruce_Up procedure goes through the Henry array data and sorts out the local and global data flow paths. It does this through the use of the LOCAL NAME function. This function searches either the Henry array for a particular procedure name or the package and appropriate procedure's declaration sections for the variable name in question. Its purpose is to sort out the local procedure or function calls from the global data structure manipulations. It cannot completely solve this problem but defers final resolution to the Calculation procedure.

The Calculate_Metric procedure will again process the Henry array data looking for the final resolution to local procedure or function calls as opposed to a

global data structure manipulation. It proceeds in small increments to finally arrive at the complexity metric calculation and a global data flow calculation. The complexity metric number is arrived at by first considering all the in, out, in-out formal parameters to calculate the fan-in and fan-out numbers. After the initial cut the fan-in, fan-out numbers are incremented upward by the numbers of identified procedure actual parameters that feed these formal parameters and then by the the Transitivity_In and Transitivity_Out functions.

An example of this process would be for procedure A with formal parameters X, Y. First process parameters X and Y for their explicit type adding 1 to fan-in if its an input parameter or 1 to fan-out if its an output parameter. Next process all the assignment expressions looking for a modification of the formal parameter. If procedure A modifies parameter X prior to a call to another function increment the fan-out count by the number of statements after the assignment delimiter. Then go through an analysis of transitivity incrementing fan-in or fan-out accordingly. Finally, call up the appropriate record of Henry_Line_count and calculate the length of the procedure or function in question.

The equation that the process is working toward solving is:

Complexity = length * (fan-in * fan-out) ** 2

This equation represents the local data flows within the analyzed procedure. Sallie Henry set the exponent of the bracketed expression to 2 because of her experience with operating system code analysis. This program will continue with

this number until enough data can be compiled to support a change. Once this calculation is done then the global data flows are analyzed.

The global flows are arrived at by first eliminating all other possibilities. Then the remaining choices have to be foreign data flows. This process is started in the Spruce_Up procedure and completed within the Metric_Calculation procedure. The process is used to find whether the data structure is being read from or written to or both.

The equation that the analysis is striving to solve is:

$$\text{Global flow} = \text{write} * (\text{read} + \text{read-write}) + \text{read-write} * (\text{read} + \text{read-write} - 1)$$

This equation represents how and by what means the global data structures are manipulated. The global data analysis procedure goes across procedure or function boundaries whereas the previous complexity metric calculations remain within the particular procedure or function under scrutiny. This across- the-border calculation is accomplished through the text file that is discussed next.

Within the calculation procedure the initial entries for the display package are started. This amounts to constructing a text file of descriptive terms and indices of merit for output in the Display_Package. It also provide a temporary storage bank for the global data information. This across-boundary analysis of

global flows was necessary because of the implications of not being able to detect the difference within the Ada code of an access to a data structure or a function call to a "withed" package.

In summary, the Analysis package is a series of analytical steps. The purpose of these steps is to arrive at the complexity and global flow metrics. These indices and additional data are stored in a text file for output to the user within the Display package.

D. HENRY DISPLAY PACKAGE

The Henry Display package is the user interface portion of the metric program. It provides the user with four different aspects of viewing the analyzed data. The purpose of this package is to show the user the data flow characteristics of the particular parsed input program. The output data will be the fan-in, fan-out, length, complexity, and four global flow numbers. These numbers can be presented in a listing format, viewed with a help file of informative paragraphs or compared by means of the other portions of the analyzed package to gain a relative sense of merit.

The procedures that comprise the Display package are LIST_METRIC_DATA and WRITE_RELATIVE_DATA and GRAPH_RELATIVE. The LIST_METRIC_DATA procedure will output the data file compiled while in the calculation portion of the previously discussed package. It will be a straight listing of information that will be grouped by each

element in the calculation of the complexity or global flow numbers, such as all procedures are grouped under the head of FAN-IN. The purpose of this listing is to show each procedure or functions component figure in the calculation of the final complexity and global figures. If the programmer is in a compile, test, edit, recompile mode of operation this will provide a spotlight on where to improve the data flow "choke- points". These data flow critical points will be seen as either high global flow or high complexity numbers. In short, the LIST_METRIC_DATA is designed for a more sophisticated programmer wishing to edit-and-run and see the results of particular programming style changes.

The WRITE_RELATIVE_DATA display will provide the same format of data but the numbers will have been normalized. Accompanying each number set will be a short narrative keyed to the relationships of the particular numbers. That is, if the user sees a complexity number of 125 beside the procedure X he will be provided with an explanation that that number is not too far out of line in comparison to the other procedures or functions analyzed within this package. The purpose of this approach is to normalize the output numbers to provide a relative comparison for a more user friendly approach to the mysterious metric number generation.

There is an additional procedure within the Display package that provides a complete listing of the raw input data. This procedure will most likely be of no use to anyone except those programmers who are extremely interested in the factors that lead to the particular numbers presented.

The final package for viewing the data is the graphical presentation module. It takes the relative data and manipulates the floating point numbers to achieve a bar chart display.

In summary, the Display package will provide the user with information in a variety of formats so that he can reach a conclusion from relative merit or absolute input numbers. The data flow numbers will point out the critical data flow points within a procedure so that the programmer can better see where to improve or expend the most effort. The purpose of the output data is to show the user where to improve, not how to improve.

E. TESTING

The testing of the design was conducted as the modules were being built and at the integration step prior to the final product. This was accomplished through the use of test harnesses that simulated the particular module's inputs and through test input programs that were hand analyzed to verify the metric's outputs.

The testing of the Henry package was accomplished by gradually building a more thorough test harness as each previous test was successful. The final test harness encompassed over 200 input records that simulated a myriad of token record inputs. The testing of the Henry module presented some difficulty because it is so intimately tied to the parsers. This was overcome by simulating the Bypass Support package as a partial input and the test harness as the balance of

the test vehicle. The package performed well within the test harness and functions adequately within the context of the entire program.

The testing of the analysis package of the Henry metric again was an iterative build of the harness. The testing accomplished after all the Henry metric packages were integrated was accomplished on the same group of programs provided by the NWC programmers to test the Halstead metric integration. The harness testing was comprised of a 50 step program that simulated a package with three independent procedures/functions utilized within its scope. There were intentional references outside the scope of the test harness package to determine if the global call detection scheme functioned properly. In all, the test harness exercised every possible data flow scheme analyzable by the Henry metric including one that would be a missed call. The analysis package performs adequately within the scope of the harness. Testing revealed that the code within the analysis phase was non- reentrant which required a the use of a boolean to define the status of the call to the package. This boolean will protect the data structure and effectively make the code reentrant.

The display module was tested with the same driver harness as the analysis package. The results were used to fine tune the package and to debug the problems. The process used was to call the analysis package from the display package and drive the display package with the test harness. This is also how the integrated program performs. The results were adequate from the stand point of the test harness but need some refinement when using the whole program.

The summary of testing would be extensive use of complicated test harnesses. Since a test harness has to simulate all the inputs to a module that the tested code could possibly see during integration, they are difficult to build much less debug. The debugging problem comes from the question 'is it the code or is it the harness?'. The test harness approach is quite fruitful from two orientations: (1) it forces the programmer into a thoroughly understanding his code and (2) the harness construction will lead the programmer into optimizing his code. Why doesn't the programmer already understand his code? He does but the ramifications of a certain approach does not come surface until the design of a test harness is considered. The optimization is driven by the need to get accurate, fast results so that the troubleshoot-repair-compile-troubleshoot regimen can proceed fairly rapidly. This is a real concern with the tremendous

IV. CONCLUSIONS

A. IMPLEMENTATION

The Henry metric was implemented in as modularized a fashion as possible. The intent was to allow for improvements through a more thorough use of the parser's information in AdaMeasure's first revision. Also, certain aspects of the Henry metric were not implemented, but it is now felt that they would add depth to the analysis process. In particular, the first change should be that the Information Flow Structure be added. This tree-like structure will allow the analysis of hidden calls but will still not detect the missed call problem discussed earlier. The missed call problem will most likely only be solved through the use of the Program Counter Register, but this approach defeats the idea of a high level language. The final improvement would be to add analysis of the "withed" packages so that an interface table could be constructed.

The program was incorporated into the previous work by Neider and Fairbanks. Their work was extensive and deserves favorable mention because it made the implementation of the Henry metric considerably easier. The output of the program is still in need of sophistication and improvement. In particular, two improvements are needed: (1) explaining the theory behind the metric and (2) conveying the ideas to the user. For an example, a high global data flow indicates an overworked global data structure. What should the metric present to the user?

The average programmer might not see the relevance of this and would miss the indication that a critical point in the data flow should probably be revised.

B. THE FUTURE OF METRICS

Metrics are tools. They point out areas of weakness. The metric will show a direction to proceed even in the absence of an absolute answer as to the correctness of the analyzed code.

The importance of metrics will grow as the size of programs grow. We do not know how important metrics will become but it does seem clear that there is a need for something that helps improve code quality and is fairly painless to use. The emphasis on "good" code will continue to be in the forefront of the Armed Service's concerns because of their intense involvement in real time embedded programs. These programs present a real challenge for incorporation of changes, improvements or any other form of maintenance programming. The purpose of metrics in this environment would be to point the way to good modularized design.

The metric should be part of the test scenario besides being an integral member of the life cycle of the program. The metric will force quality control without the painful process of formal inspections. The formal process has its place but the metric tool could perform more than the inspection. The metric tool should be incorporated into the test cycle as a meter of improvement. This immediate feedback to the programmer will be beneficial. The manager could

also use the absolute number as a goal for acceptance. This approach will provide the manager with the data needed at decision points in the life cycle of a program. The absolute number could also be tied to the program throughout its life as a measure of improvement or degradation over time. The uses are many, as the reader can see. The importance of the metric cannot be overstated when the future holds programs that will span millions of lines of code.

Metrics are important. They hold out the hope of an automated tool that will guide, interpret, and assess progress for programmers and management alike. I hope that the work of this metric will assist in advancing metrics and the use of the Ada language.

APPENDIX A: INFORMATION FLOW MECHANISMS

MECHANISMS FOR INFORMATION FLOW ANALYSIS

As Sallie Henry so succintly states:

The information flow analysis takes place in three phases. The first phase involves generating a set of relations indicating the flow of information through input parameters, output parameters, returned values from functions and data structures.

General Format of a Relation

The generation of relations is first prefaced with a quick review of relational format.

L - R1, R2 ...Rcount:

Where L may be in any one of the following forms:

1. P.DS P is the name of a procedure and DS the data structure.
2. P.O. P is the procedure name and O is the return value.
3. P.j.O P is the procedure j is an integer representing the formal parameter position, and O is the jth Output parameter.
4. P.j.I P is the procedure. j is an integer representing the jth parameter, and I is the jth input parameter.

Ri may be in one of the following forms:

1. S.DS S is a procedure name and DS is the name of a data structure.
2. S.O S is a procedure name and O is the returned value.
3. S.j.I S is the procedure name, j is the jth parameter and I is the jth input parameter.
4. S.j.O S is the procedure name, j is an integer representing the jth parameter in the list and O is the output parameter.
5. S.null S is the procedure name, null represents no data

6. S.cons. S is the procedure name and constant a value used within S.
7. S.error S is the procedure name and error represents an invalid flow of information through procedure S.

RULES

1. L is of the form P.DS then
this form is used only to generate the relations from procedure P that updates DS with Ri.
2. L is of the form P.O then
This is used only in generating the relations from procedure P that produce an output.
3. L is of the form P.j.O then
This is used when generating the relations that produce an input of the jth parameter in the procedure's formal parameter list. There must be a unique relation for each of P's parameters.
4. L is of the form P.j.I then
This is used when generating the relations for procedure P that produce an input for the jth parameter. Another procedure T calls P to indicate that the jth parameter of P receives the input update.
5. Ri is of the form S.DS then
Procedure S reads information from DS this format is used to indicate a read only.
6. Ri is of the form S.O then
Relations are generated that come from procedure T that are return values to T from S.
7. Ri is of the form S.j.I then
For generating relations for procedure S that indicates S's jth input parameter passes information to L.
8. Ri is of the form S.cons.
Then S causes a constant number or string to flow to L.
9. Ri is of the form S.Null then
This is used to indicate when S does not update a parameter, that is, the parameter was strictly input only.
10. Ri is of the form S.error then
S calls T and one of the parameters to T is an output only thus if S attempts to input a value this would be an error.

ANALYSIS OF CALLS

The following two procedures X and Y, exhibit all possible calling structures in the light of information flow analysis. NP stands for not possible, NC for no calls and the numbers beneath are used for later reference.

The pairs 1, 3, 5, 7, 13, and 15 cannot appear in a flow of data path because for DS's the only assignment and reads allowed are from procedures or functions. The other not possibles stem from input parameters not flowing into DS's and not flowing into output parameters. Entries 2 and 4 indicate X calling Y, receiving information from Y and using this information to update a DS. The rest of the possibilities can be reasoned in like manner except entries 10 and 12, which represent calls via a third procedure. Here procedure Z calls Y and passes the returned value from Y to X. This represents a no call between X and Y but there is a data flow.

TABLE 1.

LOCAL CALL TABLE

	X.DS	X.O	X.K.I	X.K.O
Y.DS	NP 1	NP 5	Y calls X 9	NP 13
Y.O	X calls Y 2	X calls Y 6	NC 10	X calls Y 14
Y.K.I	NP 3	NP 7	Y calls X 11	NP 15
Y.K.O	X calls Y 4	X calls Y 8	NC 12	X calls Y 16

All data flows from the highest calling structure, eventually being deposited in the data structures. Analysis of the table's data confirms this premise.

MEMORYLESS PROCEDURES

Some procedures keep no record of their data passing or the data supplied. These procedures are used to do housekeeping for memory management, for example, but their analysis for data flow would produce a false amount of data transactions. Another area that these procedures appear in are arithmetic operations that are sometimes duplicated in hardware such as double precision math etc. This discussion leads to the problem that these procedures would be difficult to discern in an automated process. That is, if memoryless procedures are not to be considered in data flow analysis some form of human decision making is required. It should be noted that this is another premise that the automation of the Henry metric is based on. The absolute numbers for the Henry metrics would

TABLE 2.

GLOBAL CALL TABLE

	X.DS	X.O	X.K.I	X.K.O
Y.DS	NP 1	NP 5	Y flows X 9	NP 13
Y.O	Y flows X 2	Y flows X 6	Y flows X 10	Y flows X 14
Y.K.I	NP 3	NP 7	Y flows X 11	NP 15
Y.K.O	Y flows X 4	Y flows X 8	Y flows X 12	Y flows X 16

be inflated if memoryless procedures are not eliminated from the analysis. In short a memoryless procedure should be removed from the code to be analyzed if a more accurate assessment or if the absolute numbers produced are being used for a comparative study.

APPENDIX B: HENRY METRIC CODE

```
--*******************************************************
--
-- TITLE:         AN ADA SOFTWARE METRIC
-- MODULE NAME:   PACKAGE HENRY_GLOBAL
-- DATE CREATED:  09 MAY 87
-- LAST MODIFIED: 19 MAY 87
--
-- AUTHOR:        LCDR PAUL M. HERZIG
--
--
-- DESCRIPTION: This package contains the data declarations
--    and basic procedures used throughout the Henry metric.
--
--*********************************************************

with GLOBAL, TEXT_IO;
use  GLOBAL, TEXT_IO;

package HENRY_GLOBAL is

 package INTEGER_IO is new TEXT_IO.INTEGER_IO(INTEGER);
 use INTEGER_IO;

 package REAL_IO is new TEXT_IO.FLOAT_IO(FLOAT);
 use REAL_IO;

--Real_IO produces floating point output

MAX_ARRAY_SIZE  : constant integer   := 50;
MAX_LINE_SIZE   : constant integer   := 76;
DUMMY9s         : constant integer   := 9999;
NULL_CHAR       : constant character := ' ';

--DUMMY9s are used for false data input to the line length calculations

type DECLARED_TYPE is    (BLANK, LOCAL_DECLARE, GLOBAL_DECLARE);

type ACTION_TYPE  is   (UNDEFINED,
                        HENRY_HEAD_NODE,
                        PACKAGE_TYPE,
                        PROCEDURE_TYPE,
                        FUNCTION_TYPE,
                        PARAM_TYPE,
                        ASSIGN_TYPE,
                        IDENT_TYPE,
                        DATA_STRUCTURE,
                        FUNCALL_OR_DS,
```

```
                    PROCALL_OR_DS,
                    END_PARAM_DECLARE,
                    END_ACTUAL_PARAM,
                    END_DECLARATIONS,
                    END_ASSIGN_TYPE,
                    END_PACKAGE_DECLARE,
                    END_PACKAGE_TYPE,
                    END_FUNCTION_TYPE,
                    END_PROCEDURE_CALL);

type PARAM_CLASS    is (NONE, IN_TYPE, OUT_TYPE, IN_OUT_TYPE,
                 ACTUAL_PARAM);
subtype FORMAL_PARAM_CLASS is PARAM_CLASS range IN_TYPE..IN_OUT_TYPE;
subtype LEXEME_TYPE  is  string (1.. MAX_LINE_SIZE);
subtype END_UNITS   is  ACTION_TYPE range
                 END_FUNCTION_TYPE..END_PROCEDURE_CALL;

--Declared, action and parameter classes or types are used
--in the Henry record data collection process

type HENRY_RECORD;
type POINTER is access HENRY_RECORD;
type HENRY_RECORD is record
   IDENTITY    : DECLARED_TYPE;
   NOMEN       : LEXEME_TYPE;
   TYPE_DEFINE : ACTION_TYPE;
   PARAM_TYPE  : PARAM_CLASS;
   NEXT1       : POINTER;
 end record;

--Henry record is the workhorse storage medium


type HENRY_LINE_COUNT_RECORD;
type LINE_POINTER is access HENRY_LINE_COUNT_RECORD;
type HENRY_LINE_COUNT_RECORD is record
   ID_NAME          : LEXEME_TYPE;
   START_COUNT      : INTEGER;
   STOP_COUNT       : INTEGER;
   NEXT_REC         : LINE_POINTER;
 end record;

--Henry line count record is used to claculate the length of procedures
--or functions

type HENRY_DATA is record
 NAME_OF_DATA         : LEXEME_TYPE;
 BEGIN_POINTER        : POINTER;
 LINE_LENGTH_POINTER  : LINE_POINTER;
end record;

--Henry data records are used to delineate the functions and procedures
```

```
--for easier data calculations

type HENRY_DATA_ARRAY is array (1..MAX_ARRAY_SIZE) of HENRY_DATA;

type OUTPUT_DATA is record
  TYPE_OF          : ACTION_TYPE := UNDEFINED;
  NAME_OF          : LEXEME_TYPE;
  TYPE_FAN_IN      : FLOAT := 0.0;
  TYPE_FAN_OUT     : FLOAT := 0.0;
  TYPE_COMPLEXITY  : FLOAT := 0.0;
  TYPE_READ        : FLOAT := 0.0;
  TYPE_WRITE       : FLOAT := 0.0;
  TYPE_READ_WRITE  : FLOAT := 0.0;
  TYPE_FLOW        : FLOAT := 0.0;
  CODE_LENGTH      : INTEGER := 0;
end record;


--Output data records hold the final calculation numbers for storage into
--an output/input file

type OUTPUT_ARRAY is array (1..MAX_ARRAY_SIZE) of OUTPUT_DATA;

NEXT_HEN, LAST_RECORD, NEW_RECORD,
HEAD, NAME_POINTER                 : POINTER;
HENRY_ARRAY                        : HENRY_DATA_ARRAY;
HENRY_LINE_COUNT                   : integer := 0;
OUT_PUT_DATA                       : OUTPUT_ARRAY;
LINE_COUNT_RECORD                  : HENRY_LINE_COUNT_RECORD;
HEAD_LINE, NEXT_LINE, LAST_LINE    : LINE_POINTER;
PACKAGE_BODY_DECLARE,
ASSIGN_MARKER,
GLOBAL_MARKER,
NAME_TAIL_SET,
ASSIGN_STATEMENT,
FUNCTION_PARAM_DECLARE,
FORMAL_PARAM_DECLARE               : BOOLEAN := FALSE;
FIRST_HENRY_CALL                   : BOOLEAN := TRUE;
DUMMY_LEXEME                       : LEXEME_TYPE;

procedure CREATE_NODE(NEW_NODE, LAST_RECORD : in out POINTER);
procedure CREATE_LINE_COUNT_NODE(NEXT_LINE,
                 LAST_LINE  : in out LINE_POINTER);
procedure INITIALIZE_HENRY(HEAD      : in out POINTER;
              HEAD_LINE : in out LINE_POINTER);



procedure CLEAR_HENRY_LEXEME(HENRY_LEXEME   : in out LEXEME_TYPE);

end HENRY_GLOBAL;
-------------------------------------------------------------
```

```
---------------------------------------------------------------

package body HENRY_GLOBAL is
---------------------------------------------------------------
--procedure creates Henry record nodes for data storage

procedure CREATE_NODE(NEW_NODE, LAST_RECORD : in out POINTER) is

TEMP_POINTER : POINTER;

begin
  put(result_file, "in create henry node"); new_line(result_file);
  TEMP_POINTER            := new HENRY_RECORD;
  TEMP_POINTER.IDENTITY   := BLANK;
  for I in 1..MAX_LINE_SIZE loop
   TEMP_POINTER.NOMEN(I)  := NULL_CHAR;
  end loop;
  TEMP_POINTER.TYPE_DEFINE := UNDEFINED;
  TEMP_POINTER.PARAM_TYPE  := NONE;
  NEW_NODE.NEXT1          := TEMP_POINTER;
  LAST_RECORD             := NEW_NODE;
  NEW_NODE                := TEMP_POINTER;
end CREATE_NODE;

---------------------------------------------------------------
--creates line count nodes to hold the length data for each procedure or
--function

procedure CREATE_LINE_COUNT_NODE(NEXT_LINE,
                          LAST_LINE   : in out LINE_POINTER) is

TEMP_POINTER : LINE_POINTER;

begin
  put(result_file, "in henry create line node"); new_line(result_file);
  TEMP_POINTER            := new HENRY_LINE_COUNT_RECORD;
  for I in 1..MAX_LINE_SIZE loop
   TEMP_POINTER.ID_NAME(I) := NULL_CHAR;
  end loop;
  TEMP_POINTER.START_COUNT := DUMMY9s;
  TEMP_POINTER.STOP_COUNT  := DUMMY9s;
  NEXT_LINE.NEXT_REC      := TEMP_POINTER;
  LAST_LINE               := NEXT_LINE;
  NEXT_LINE               := TEMP_POINTER;
end CREATE_LINE_COUNT_NODE;

---------------------------------------------------------------
--sets all of the variables to their initial values besides
--creating the first Henry record and line count record

procedure INITIALIZE_HENRY(HEAD      : in out POINTER;
                   HEAD_LINE : in out LINE_POINTER) is
```

```
HEAD_STRING : STRING(1..9)  := "HEAD NODE";
SIZE                   : INTEGER := 9;

begin
  CREATE(HENRY_FILE, out_file, HENRY_FILE_NAME);
  put(HENRY_FILE, "in INITIALIZE HENRY"); new_line(HENRY_FILE);
  CREATE(HENRY_OUT, out_file, HENRY_OUT_NAME);
  HEAD              := new HENRY_RECORD;
  HEAD.NOMEN(1..SIZE) := HEAD_STRING;
  HEAD.IDENTITY      := BLANK;
  HEAD.TYPE_DEFINE   := HENRY_HEAD_NODE;
  HEAD.PARAM_TYPE    := NONE;
  NEXT_HEN          := HEAD;
  CREATE_NODE(NEXT_HEN, LAST_RECORD);
  HENRY_LINE_COUNT   := 0;
  DUMMY_LEXEME(1)    := NULL_CHAR;
  HEAD_LINE          := new HENRY_LINE_COUNT_RECORD;
  HEAD_LINE.ID_NAME(1..SIZE)  := HEAD_STRING;
  HEAD_LINE.START_COUNT      := DUMMY9s;
  HEAD_LINE.STOP_COUNT       := DUMMY9s;
  NEXT_LINE              := HEAD_LINE;
  CREATE_LINE_COUNT_NODE(NEXT_LINE, LAST_LINE);
end INITIALIZE_HENRY;

-------------------------------------------------------------

--clears the input string to null characters

procedure CLEAR_HENRY_LEXEME(HENRY_LEXEME : in out LEXEME_TYPE) is

begin
put(HENRY_FILE, "IN CLEAR HENRY LEXEME"); NEW_LINE(HENRY_FILE);
FOR I in 1 ..MAX_LINE_SIZE loop
 HENRY_LEXEME(I) := NULL_CHAR;
end loop;
END CLEAR_HENRY_LEXEME;

END HENRY_GLOBAL;



--***********************************************************
--
-- TITLE:          AN ADA SOFTWARE METRIC
--
-- MODULE NAME:   PACKAGE HENRY_METRIC
-- DATE CREATED:  06 APR 87
-- LAST MODIFIED: 15 MAY 87
--
-- AUTHORS        LCDR PAUL M  HERZIG
--
```

```
--
-- DESCRIPTION:  This package contains the Henry metric data
--          collection and program control routines.
--
--*****************************************************************

with GLOBAL, HENRY_GLOBAL, HENRY_ANALYSIS, HENRY_DISPLAY, TEXT_IO;
use  GLOBAL, HENRY_GLOBAL, HENRY_ANALYSIS, HENRY_DISPLAY, TEXT_IO;

package HENRY is



procedure WRITE_HENRY_DATA(ID      : in DECLARED_TYPE := BLANK;
                   IN_NAME : in LEXEME_TYPE := DUMMY_LEXEME;
                   DEFINE  : in ACTION_TYPE := UNDEFINED;
                   PARAM   : in PARAM_CLASS := NONE;
                   LINK    : in POINTER);


procedure UPDATE_LINE_COUNT;

procedure WRITE_LINE_COUNT(IN_NAME : in LEXEME_TYPE:= DUMMY_LEXEME;
                    FIRST_COUNT : in INTEGER := DUMMY9s;
                    LAST_COUNT  : in INTEGER := DUMMY9s;
                    PTR         : in LINE_POINTER);
end HENRY;

-----------------------------------------------------------------

-----------------------------------------------------------------

package body HENRY is



-----------------------------------------------------------------


--produces the written data records from the parser inputs
--data is only written if it is something other than the
--null settings

procedure WRITE HENRY DATA(ID      in DECLARED TYPE   BLANK
                   IN NAME : in LEXEME TYPE   DUMMY LEXEME
                   DEFINE   in ACTION TYPE   UNDEFINED
                   PARAM    in PARAM CLASS   NONE
                   LINK     in POINTER) IS
begin
put(result file "in write henry data") new line(result file)
 If ID          BLANK then
   LINK IDENTITY      ID
```

```
    case ID is
      when LOCAL_DECLARE    => put(RESULT_FILE, "Local declare");
      when GLOBAL_DECLARE   => put(RESULT_FILE, "Global declare");
      when others           => put(RESULT_FILE, "Undeclared");
    end case;
 else put(RESULT_FILE, "NO DECLARATION");
 end if;
 new_line(RESULT_FILE);
 If IN_NAME(i) /= NULL_CHAR then
   LINK.NOMEN(1..MAX_LINE_SIZE) := IN_NAME(1..MAX_LINE_SIZE);
   PUT(RESULT_FILE, IN_NAME);
 ELSE PUT(RESULT_FILE, "NO NAME");
 end if;
 new_line(RESULT_FILE);
 If DEFINE /= UNDEFINED then
   LINK.TYPE_DEFINE := DEFINE;
case DEFINE is
when UNDEFINED            => put(RESULT_FILE, "Undefined");
when HENRY_HEAD_NODE      => put(RESULT_FILE, "Henry Head Node");
when PACKAGE_TYPE         => put(RESULT_FILE, "Package declaration");
when PROCEDURE_TYPE       => put(RESULT_FILE, "Procedure declaration");
when FUNCTION_TYPE        => put(RESULT_FILE, "Function declaration");
when PARAM_TYPE           => put(RESULT_FILE, "Parameter declaration");
when ASSIGN_TYPE          => put(RESULT_FILE, "Assignment delimiter");
when IDENT_TYPE           => put(RESULT_FILE, "Identifier");
when DATA_STRUCTURE       => put(RESULT_FILE, "Data structure descriptor");
when FUNCALL_OR_DS        => put(RESULT_FILE, "Function or data descriptor");
when PROCALL_OR_DS        => put(RESULT_FILE, "Procedure or data descriptor");
when END_PARAM_DECLARE    => put(RESULT_FILE, "End parameter delimiter");
when END_ACTUAL_PARAM     => put(RESULT_FILE, "End actual parameter delimiter");
when END_DECLARATIONS     => put(RESULT_FILE, "End declaration delimiter");
when END_ASSIGN_TYPE      => put(RESULT_FILE, "End assignment statement delimiter");
when END_PACKAGE_DECLARE  => put(RESULT_FILE, "End package declaration delimiter");
when END_PACKAGE_TYPE     => put(RESULT_FILE, "End package delimiter");
when END_FUNCTION_TYPE    => put(RESULT_FILE, "End function delimiter");
when END_PROCEDURE_CALL   => put(RESULT_FILE, "End procedure delimiter");
when others               => put(RESULT_FILE, "Unknown");
end case;
new_line(RESULT_FILE);
 end if;
 If PARAM /= NONE then
   LINK.PARAM_TYPE := PARAM;
   CASE PARAM IS
   WHEN IN_TYPE     => PUT(RESULT_FILE, "IN PARAM");
   WHEN OUT_TYPE    => PUT(RESULT_FILE, "OUT PARAM");
   WHEN IN_OUT_TYPE => PUT(RESULT_FILE, "IN OUT PARAM");
   WHEN OTHERS      => PUT(RESULT_FILE, "NONE");
  END CASE;
 end if;
 new_line(RESULT_FILE);
end WRITE_HENRY_DATA;
```

```
--------------------------------------------------------------

--increments the line count for eventual inclusion into
--the calculation of a particular procedures total length
--the length number is used in the complexity calculation

procedure UPDATE_LINE_COUNT is



begin
put(result_file, "in update line count"); new_line(result_file);
 if not FORMAL_PARAM_DECLARE then
  HENRY_LINE_COUNT := HENRY_LINE_COUNT + 1;
 end if;
end UPDATE_LINE_COUNT;



--------------------------------------------------------------


--produces the records to hold the line count information
--the records are not initially tied to a particular procedure
--but are a parallel data structure until in the Hen_anal.pkg
--where they are linked to the procedure that they hold the
--data for

procedure WRITE_LINE_COUNT(IN_NAME : in LEXEME_TYPE := DUMMY_LEXEME;
                     FIRST_COUNT : in INTEGER := DUMMY9s;
                     LAST_COUNT  : in INTEGER := DUMMY9s;
                     PTR         : in LINE_POINTER) IS

begin
 put(HENRY_FILE, "in WRITE_LINE_COUNT"); new_line(HENRY_FILE);
put(result_file, "in write line count"); new_line(result_file);
 If IN_NAME(1) /= NULL_CHAR then
          PTR.ID_NAME(1..MAX_LINE_SIZE) := IN_NAME; end if;
 If FIRST_COUNT /= DUMMY9s then PTR.START_COUNT := FIRST_COUNT; end if;
 If LAST_COUNT /= DUMMY9s then PTR.STOP_COUNT := LAST_COUNT; end if;

end WRITE_LINE_COUNT;

end HENRY;




--************************************************************
--
-- TITLE          AN ADA SOFTWARE METRIC
--
-- MODULE NAME    PACKAGE HENRY_ANALYSIS
```

```
-- DATE CREATED   29 APR 87
-- LAST MODIFIED  29 MAY 87
--
-- AUTHOR         LCDR PAUL M HERZIG
--
-- DESCRIPTION: This package contains the analysis functions
--    required to identify each data flow in the Henry metric
--
--*****************************************************************

with GLOBAL, GLOBAL_PARSER, BYPASS_SUPPORT_FUNCTIONS, HENRY_GLOBAL, TEXT_IO;
use GLOBAL, GLOBAL_PARSER, BYPASS_SUPPORT_FUNCTIONS, HENRY_GLOBAL, TEXT_IO;

package HENRY_ANALYSIS is

 package NEW_INTEGER_IO is new TEXT_IO.INTEGER_IO(integer);
 use NEW_INTEGER_IO;

 package REAL_IO is new TEXT_IO.FLOAT_IO(float);
 use REAL_IO;

PROC_FUNC_COUNT : INTEGER := 0;
INDEX           : INTEGER;
NAME_POINTER    : POINTER;

--PROC_FUNC_COUNT is the total number of procedures and functions in the
--analyzed package


type SELECTOR_TYPE is (PROCEDURE_FIND, FUNCTION_FIND,
          VARIABLE_FIND);
procedure CLEAN_UP_HENRY_DATA(HEAD : in POINTER);
procedure SET_UP_HENRY_ARRAY(HEAD : in POINTER;
                HEAD_LINE   : in LINE_POINTER);

procedure SPRUCE_UP_HENRY_DATA;
function LOCAL_NAME(NAME_POINTER    : in POINTER;
             SELECTOR       : in SELECTOR_TYPE;
             INDEX          : in INTEGER )
              return BOOLEAN;
function CALCULATE_LINE_COUNT(WORK_LINE : LINE_POINTER)
                 return INTEGER;
function FIND_STRING_SIZE(IN_STRING : LEXEME_TYPE) RETURN INTEGER;
function TRANSITIVITY_IN(IN_NAME : LEXEME_TYPE;
              BEGIN_LOOP, STOP_LOOP : POINTER)
              RETURN FLOAT;
function TRANSITIVITY_OUT(IN_NAME : LEXEME_TYPE;
              TOP      : POINTER)
              RETURN FLOAT;
procedure CALCULATE_METRIC(HEAD        : in POINTER;
              HEAD_LINE    : in LINE_POINTER);
```

```
end HENRY_ANALYSIS;
----------------------------------------------------------------

----------------------------------------------------------------
package body HENRY_ANALYSIS is

----------------------------------------------------------------
--starts the process of setting up the raw Henry records into
--decipherable data. it also counts the numbers of procedures
--functions and fills in empty parameter type fields in the
--Henry records

procedure CLEAN_UP_HENRY_DATA(HEAD : IN POINTER) is

TEMP, TOP, BOTTOM : POINTER;

begin
 put(HENRY_FILE, "in CLEAN_UP_HENRY"); new_line(HENRY_FILE);
 CLEARSCREEN;
 put("Processing Henry data records ... please wait");
 TOP    := HEAD;
 BOTTOM := TOP.NEXT1;

-- move past package declarations

 LOOP
  EXIT WHEN TOP.TYPE_DEFINE = END_PACKAGE_DECLARE;
  TOP    := BOTTOM;
  BOTTOM := TOP.NEXT1;
 END LOOP;

--count the number of procedures/functions

 LOOP
   EXIT WHEN BOTTOM.TYPE_DEFINE = END_PACKAGE_TYPE;
   if (BOTTOM.TYPE_DEFINE = PROCEDURE_TYPE) or
     (BOTTOM.TYPE_DEFINE = FUNCTION_TYPE) then
    PROC_FUNC_COUNT := PROC_FUNC_COUNT + 1;
   end if;

   TEMP   := BOTTOM;
   BOTTOM := TEMP.NEXT1;
 end loop;
 BOTTOM := TOP;

--ensure all parameter records have a type defined

   FOR I in 1..PROC_FUNC_COUNT LOOP
     LOOP
       EXIT WHEN (TOP.TYPE_DEFINE = PROCEDURE_TYPE) OR
             (TOP.TYPE_DEFINE = FUNCTION_TYPE);
      TOP := BOTTOM.NEXT1;
```

```
    BOTTOM     TOP.
  END LOOP.
  TEMP  -  TOP NEXT1.
  if TEMP TYPE DEFINE    PARAM TYPE AND
    TOP TYPE DEFINE      FUNCTION TYPE then
    LOOP
      EXIT WHEN TEMP TYPE DEFINE    END PARAM DECLARE.
      if TEMP PARAM TYPE NOT IN FORMAL PARAM CLASS THEN
       LOOP
        EXIT WHEN (TEMP PARAM TYPE    IN TYPE) OR
                (TEMP PARAM TYPE    OUT TYPE) OR
                (TEMP PARAM TYPE    IN OUT TYPE);
        BOTTOM     TEMP.
        TEMP      BOTTOM NEXT1.
       END LOOP.
       BOTTOM     TEMP
       TEMP      TOP NEXT1
       TOP       TEMP.
       LOOP
        EXIT WHEN (TOP PARAM TYPE    IN TYPE) OR
                (TOP PARAM TYPE    OUT TYPE) OR
                (TOP PARAM TYPE    IN OUT TYPE).
        TEMP PARAM TYPE    BOTTOM PARAM TYPE
        TEMP      TOP NEXT1
        TOP       TEMP
       END LOOP
      else
       TOP     TEMP
       BOTTOM     TEMP
      end if
      TEMP     TOP NEXT1
    END LOOP

--functions usually invoke the default in type parameter
--insert this type if it is not defined

  elsif TOP TYPE DEFINE    FUNCTION TYPE THEN
   if TEMP TYPE DEFINE    PARAM TYPE THEN
    LOOP
      EXIT WHEN TEMP TYPE DEFINE    END PARAM DECLARE
       TEMP PARAM TYPE    IN TYPE
       TEMP      BOTTOM NEXT1
       BOTTOM     TEMP
    END LOOP
   end if.
  end if
  TOP       BOTTOM NEXT1
  BOTTOM     TOP
 END LOOP    FOR LOOP
end CLEAN UP HENRY DATA
```

```
--sets up the Henry data records to mark the beginning of each
--function or procedure. it also ties the procedure line length
--records to its proper procedure function

procedure SET UP HENRY ARRAY(HEAD      : in POINTER;
                  HEAD_LINE  : in LINE POINTER) is

WORK LINE, TEMP LINE : LINE POINTER;
TEMP, TOP, BOTTOM    : POINTER;

begin
 put(HENRY FILE, "in SET UP HENRY"); new line(HENRY FILE);
 WORK LINE := HEAD LINE.NEXT REC;
 TEMP LINE := WORK LINE;
 BOTTOM := HEAD;
 TOP := BOTTOM;

--GO PAST DECLARATIONS

 LOOP
  EXIT WHEN TOP.TYPE DEFINE = END PACKAGE DECLARE;
  TOP := BOTTOM.NEXT1;
  BOTTOM := TOP;
 END LOOP;

--set up the Henry array records so that their pointers are at the
--top of each procedure or function

 FOR I in 1..PROC FUNC COUNT LOOP
   LOOP
     EXIT WHEN (TOP.TYPE DEFINE = PROCEDURE TYPE) OR
          (TOP.TYPE DEFINE = FUNCTION TYPE);
     TOP := BOTTOM.NEXT1;
     BOTTOM := TOP;
   END LOOP;
   HENRY ARRAY(I).NAME OF DATA(1..MAX LINE SIZE) :=
          TOP.NOMEN(1..MAX LINE SIZE);
   HENRY ARRAY(I).BEGIN POINTER := TOP;
   LOOP
     EXIT WHEN (BOTTOM.TYPE DEFINE = END FUNCTION TYPE) OR
          (BOTTOM.TYPE DEFINE = END PROCEDURE CALL);
     TEMP := BOTTOM.NEXT1;
     BOTTOM := TEMP;
   END LOOP;

 --set the arc count records to their related procedure function

   TOP := BOTTOM.NEXT1;
   BOTTOM := TOP;
   HENRY ARRAY(I).LINE LENGTH POINTER := WORK LINE;
   WORK LINE := TEMP LINE.NEXT REC;
   TEMP LINE := WORK LINE;
```

```
 END LOOP; --FOR LOOP
end SET UP HENRY ARRAY;
------------------------------------------------------------
--this procedure calculates the length of each procedure function
--the results are fed into line length records

function CALCULATE LINE COUNT(WORK LINE : LINE POINTER)

                    return INTEGER is

DIFFERENCE : INTEGER := 0;
I          : INTEGER;

begin
 put(HENRY FILE, "in CALCULATE LINE COUNT"); new line(HENRY FILE);
 DIFFERENCE := WORK LINE.STOP COUNT - WORK LINE.START COUNT;
 RETURN (DIFFERENCE);
end CALCULATE LINE COUNT;
------------------------------------------------------------
--this function searches for local, within a procedure, and global-local,
--within a package, for variable name matches
--it is selectable for which name the search is conducted

function LOCAL NAME(NAME POINTER   : in POINTER;
               SELECTOR      : in SELECTOR TYPE;
               INDEX         : in INTEGER )
                return BOOLEAN is

NAME SOUGHT, POINTER NAME : LEXEME TYPE;
NAME SIZE, POINTER SIZE : INTEGER := MAX LINE SIZE;
RESULT              : BOOLEAN := FALSE;
TEMP, TEMP1              : POINTER;
I                    : INTEGER := 1;

begin

put(HENRY FILE, "in LOCAL NAME"); new line(HENRY FILE);
NAME SOUGHT(1..NAME SIZE) := NAME POINTER.NOMEN(1..NAME SIZE);
CONVERT UPPER CASE(NAME SOUGHT, NAME SIZE);


if ((SELECTOR = PROCEDURE FIND) OR (SELECTOR = FUNCTION FIND))
 AND (PROC FUNC COUNT > 0) then
 LOOP
   POINTER NAME(1..POINTER SIZE) :=
   HENRY ARRAY(I).NAME OF DATA(1..POINTER SIZE);
   CONVERT UPPER CASE(POINTER NAME, POINTER SIZE);
   RESULT := (NAME SOUGHT(1..NAME SIZE) =
          POINTER NAME(1..POINTER SIZE));
   EXIT WHEN (I = PROC FUNC COUNT) OR (RESULT);
   I := I + 1;
 END LOOP;
```

```
I := 1;

--if it is a variable name search first within the package
--declarations. next within the procedure declarations

elsif SELECTOR = VARIABLE_FIND then
 TEMP := HEAD.NEXT1;
 LOOP
   EXIT WHEN (TEMP.TYPE_DEFINE = END_PACKAGE_DECLARE) OR
         (RESULT);
   if TEMP.TYPE_DEFINE = IDENT_TYPE then
     POINTER_NAME(1..POINTER_SIZE) := TEMP.NOMEN(1..POINTER_SIZE);
     CONVERT_UPPER_CASE(POINTER_NAME, POINTER_SIZE);
     RESULT := (NAME_SOUGHT(1..NAME_SIZE) =
          POINTER_NAME(1..POINTER_SIZE));
   end if;
   TEMP1 := TEMP.NEXT1;
   TEMP := TEMP1;
 END LOOP;

--did not find the variable within the package declarations
--search the specified procedures declarations

 if NOT RESULT then
   TEMP := HENRY_ARRAY(INDEX).BEGIN_POINTER;
   LOOP   -- DID NOT FIND NAME IN PACKAGE DECLARATIONS
     EXIT WHEN (TEMP.TYPE_DEFINE = END_DECLARATIONS) OR
           (RESULT);
     if TEMP.TYPE_DEFINE = IDENT_TYPE then
       POINTER_NAME(1..POINTER_SIZE) :=
           TEMP.NOMEN(1..POINTER_SIZE);
       CONVERT_UPPER_CASE(POINTER_NAME, POINTER_SIZE);
       RESULT := (NAME_SOUGHT(1..NAME_SIZE) =
          POINTER_NAME(1..POINTER_SIZE));
     end if;
     TEMP1 := TEMP.NEXT1;
     TEMP := TEMP1;
   END LOOP;
 end if;
end if;
RETURN (RESULT);
end LOCAL_NAME;


-------------------------------------------------------------

--finishes polishing the Henry records. the data can now be analyzed
--for local global data and starts the actual number crunching

procedure SPRUCE_UP_HENRY_DATA is
```

```
TEMP, TEMP1, TEMP2 : POINTER;

begin
 put(HENRY_FILE, "in SPRUCE_UP_HENRY"); new_line(HENRY_FILE);
 FOR I in 1..PROC_FUNC_COUNT LOOP
   TEMP1 := HENRY_ARRAY(I).BEGIN_POINTER;

--loop past parameters

   LOOP
     EXIT WHEN TEMP1.TYPE_DEFINE = END_DECLARATIONS;
     TEMP2 := TEMP1.NEXT1;
     TEMP1 := TEMP2;
   END LOOP;
   TEMP := TEMP1.NEXT1;

--first analyze identifier types (variables) for local or global
--significance. Update the record if it is not local

   LOOP --LOOK FOR IDENT_TYPES
     EXIT WHEN (TEMP.TYPE_DEFINE = END_FUNCTION_TYPE) OR
           (TEMP.TYPE_DEFINE = END_PROCEDURE_CALL);
     if TEMP.TYPE_DEFINE = IDENT_TYPE then
       if TEMP.IDENTITY = BLANK then
         if LOCAL_NAME(TEMP, VARIABLE_FIND, I) then
           TEMP.IDENTITY := LOCAL_DECLARE;
         else TEMP.IDENTITY := GLOBAL_DECLARE;
         end if;
       end if;
     end if;
     TEMP1 := TEMP.NEXT1;
     TEMP := TEMP1;

   END LOOP;

--now go through the Henry records looking for unresolved
--procedure or function calls update the Henry records
--to reflect procedure types or function types or data structures


   TEMP1 := HENRY_ARRAY(I).BEGIN_POINTER;
   TEMP := TEMP1.NEXT1;

--get past declarations

   LOOP
     EXIT WHEN TEMP.TYPE_DEFINE = END_DECLARATIONS;
     TEMP1 := TEMP.NEXT1;
     TEMP := TEMP1;
   END LOOP;
```

```
--looking for procedure or function calls

  LOOP
    EXIT WHEN (TEMP.TYPE_DEFINE = END_FUNCTION_TYPE) OR
          (TEMP.TYPE_DEFINE = END_PROCEDURE_CALL);

    if TEMP.TYPE_DEFINE = PROCALL_OR_DS then
      TEMP1 := TEMP;
      LOOP          -- MOVE PAST THE PARAMETERS
        EXIT WHEN TEMP1.TYPE_DEFINE = END_ACTUAL_PARAM;
        TEMP2 := TEMP1;
        TEMP1 := TEMP2.NEXT1;
      END LOOP;
      if (LOCAL_NAME(TEMP, PROCEDURE_FIND, I)) then
        TEMP.TYPE_DEFINE := PROCEDURE_TYPE;
      else
        TEMP2 := TEMP1.NEXT1;
        if TEMP2.TYPE_DEFINE = ASSIGN_TYPE then
          TEMP.TYPE_DEFINE := DATA_STRUCTURE;
          --IF NOT IT IS A PROCEDURE CALL ONLY
          TEMP1 := TEMP2.NEXT1;
          LOOP
          EXIT WHEN TEMP1.TYPE_DEFINE = END_ASSIGN_TYPE;
            if (TEMP1.TYPE_DEFINE = FUNCALL_OR_DS) then
              if NOT LOCAL_NAME(TEMP1, FUNCTION_FIND, I)
              then
                TEMP1.TYPE_DEFINE := DATA_STRUCTURE;
              else TEMP1.TYPE_DEFINE := FUNCTION_TYPE;
              end if;
            end if;
            TEMP2 := TEMP1;
            TEMP1 := TEMP2.NEXT1;
          END LOOP;
        else TEMP.TYPE_DEFINE := PROCEDURE_TYPE;
        end if;
      end if;

--only function calls that cannot be resolved into a local name are
--specified as data structures

    elsif TEMP.TYPE_DEFINE = FUNCALL_OR_DS then
      TEMP1 := TEMP;
      LOOP    --LOOKING FOR FUNCTIONS
      EXIT WHEN TEMP.TYPE_DEFINE = END_ASSIGN_TYPE;
        if TEMP.TYPE_DEFINE = FUNCALL_OR_DS then
          if (LOCAL_NAME(TEMP, FUNCTION_FIND, I))
          then
            TEMP.TYPE_DEFINE := FUNCTION_TYPE;
          else TEMP.TYPE_DEFINE := DATA_STRUCTURE;
          end if
        end if
        TEMP1 := TEMP.NEXT1
```

```
        TEMP := TEMP1;
      END LOOP;
    end if;
    TEMP1 := TEMP;
    TEMP := TEMP1.NEXT1;
  END LOOP; --PROCALL_OR_DS LOOP
END LOOP; -- FOR LOOP


end SPRUCE_UP_HENRY_DATA;
-------------------------------------------------------------
--this function only works for Ada language strings that identify
--a variable

function FIND_STRING_SIZE(IN_STRING : LEXEME_TYPE) RETURN INTEGER
is

SIZE : INTEGER := 0;

BEGIN
 PUT(HENRY_FILE, "IN FIND STRING SIZE"); NEW_LINE(HENRY_FILE);
  FOR I IN 1..MAX_LINE_SIZE LOOP
    IF IN_STRING(I) = NULL_CHAR THEN
      SIZE := SIZE + 1;
    END IF;
  END LOOP;
  RETURN SIZE;
END FIND_STRING_SIZE;
-------------------------------------------------------------

--transitivity is detected by searching the right hand side of
--assignment statements for a name match of the actual
--parameters from a function or procedure call

function TRANSITIVITY_IN(IN_NAME : LEXEME_TYPE;
                BEGIN_LOOP, STOP_LOOP : POINTER)
                RETURN FLOAT is

ASSIGN_MARK,
PROCEDURE_MARK : BOOLEAN := FALSE;
TRANS_COUNT    : FLOAT   := 0.0;
TEMP, TEMP1    : POINTER := BEGIN_LOOP;
T1, T2         : POINTER;
MAX            : INTEGER := MAX_LINE_SIZE;

BEGIN

--stop loop is determined by where in the parameter list you are

  LOOP
    EXIT WHEN TEMP = STOP_LOOP;
    if TEMP.TYPE_DEFINE = ASSIGN_TYPE THEN
```

```
      ASSIGN_MARK := TRUE;
    elsif TEMP.TYPE_DEFINE = END_ASSIGN_TYPE THEN
      ASSIGN_MARK := FALSE;
    end if;

--mark whether you've passed an assignment

    if (TEMP.NOMEN(1..MAX) = IN_NAME(1..MAX)) AND
      (NOT ASSIGN_MARK) THEN
      TRANS_COUNT := TRANS_COUNT + 1.0;

--if you have detected a name match count the number of assignment
--variables as transitive feed into the actual parameter
--note functions have already been calculated the same for
--data structures so skip these counts

      T1 := TEMP; T2 := T1.NEXT1;
      if (T1.TYPE_DEFINE = IDENT_TYPE) AND
        (T2.TYPE_DEFINE = ASSIGN_TYPE) then
        LOOP
         EXIT WHEN T2.TYPE_DEFINE = END_ASSIGN_TYPE;
          if T2.TYPE_DEFINE = IDENT_TYPE THEN
             TRANS_COUNT := TRANS_COUNT + 1.0;
          end if;
         T1 := T2;
         T2 := T1.NEXT1;
        END LOOP;
       end if;
    end if;
    TEMP := TEMP1.NEXT1;
    TEMP1 := TEMP;
  END LOOP;
  RETURN(TRANS_COUNT);
END TRANSITIVITY_IN;


-------------------------------------------------------------


--if detect a name match on the right hand side of an assignment
--statement have a transitive relation on this variable but
--there is no need to count the rest of the assignment
--variables becuase the most it can account for is 1

function TRANSITIVITY_OUT(IN_NAME : LEXEME_TYPE;
                  TOP     : POINTER)
                  RETURN FLOAT is

ASSIGN_MARK : BOOLEAN := FALSE;
TRANS_COUNT : FLOAT   := 0.0;
TEMP, TEMP1 : POINTER := TOP;
MAX         : INTEGER := MAX_LINE_SIZE;
```

```
BEGIN
 LOOP
   EXIT WHEN (TEMP.TYPE_DEFINE = END_PROCEDURE_CALL) OR
        (TEMP.TYPE_DEFINE = END_FUNCTION_TYPE);
   IF TEMP.TYPE_DEFINE = ASSIGN_TYPE THEN
     ASSIGN_MARK := TRUE;
   ELSIF TEMP.TYPE_DEFINE = END_ASSIGN_TYPE THEN
     ASSIGN_MARK := FALSE;
   END IF;
   IF (TEMP.NOMEN(1..MAX) = IN_NAME(1..MAX)) AND (ASSIGN_MARK)
     THEN
     TRANS_COUNT := TRANS_COUNT + 1.0;
   END IF;
   TEMP := TEMP1.NEXT1;
   TEMP1 := TEMP;
 END LOOP;
 RETURN(TRANS_COUNT);
END TRANSITIVITY_OUT;

------------------------------------------------------------

--finishes polishing the data and with the transitivity functions calculates
--the fan in   fan out of data besides the global data structures

procedure CALCULATE_METRIC (HEAD        : in POINTER;
                  HEAD_LINE   : in LINE_POINTER) is

TEMP_LINE             : LINE_POINTER;
TEMP, TOP, TEMP1, TEMP2, POINTER;
PROC_PTR,
PARAM_PTR             : POINTER;

FAN_IN, FAN_OUT       : FLOAT;
LENGTH                : INTEGER := 0;
MAX                   : INTEGER := MAX_LINE_SIZE;
CODE_EXPONENT         : INTEGER := 2;
COMPLEXITY,
GLOBAL_FLOW,
GLOBAL_READ,
GLOBAL_WRITE,
GLOBAL_READ_WRITE     : FLOAT;

--global flow represents the whole picture of global data flow
--the equation is below and encompasses both read and write to
--global data structures
--note  global data structures could be external function calls
--there is no means to determine the difference

NEW_NAME              : STRING(1..MAX_LINE_SIZE);
NAME_OF               : LEXEME_TYPE;
ASSIGN_MARK,
GLOBAL_MARK           : BOOLEAN := FALSE;
```

```
SIZE                INTEGER    MAX LINE SIZE
NEW SIZE               INTEGER    0
TEMP NAME                STRING(1 SIZE)

begin
 PUT(HENRY FILE "IN CALCULATE METRIC"   NEW LINE(HENRY FILE)


--first Henry call boolean is so that the data can be reshown

 IF FIRST HENRY CALL then
  CLEAN UP HENRY DATA(HEAD)
  SET UP HENRY ARRAY(HEAD HEAD LINE)
  SPRUCE UP HENRY DATA
  FOR I in 1 PROC FUNC COUNT LOOP
    GLOBAL READ
    GLOBAL WRITE
    GLOBAL READ WRITE
    FAN IN
    FAN OUT
    COMPLEXITY
    GLOBAL FLOW
    LENGTH
    TEMP    HENRY ARRAY(I BEGIN POINTER
    CLEAR HENRY LEXEME(TEMP NAME
    TEMP NAME(1 MAX LINE SIZE     HENRY ARRAY I NAME OF DATA
    SIZE      FIND STRING SIZE TEMP NAME
    CLEAR HENRY LEXEME(NEW NAME)
    CONVERT UPPER CASE TEMP NAME SIZE

--initialize the variables [illegible]
--calculations

    if TEMP TYPE DEFINE    PROCEDURE TYPE then
      OUT PUT DATA(I TYPE OF     PROCEDURE TYPE
      NEW SIZE               SIZE   10
      NEW NAME(1 10)              procedure
      NEW NAME(11 NEW SIZE       TEMP NAME 1 SIZE
      OUT PUT DATA I NAME OF 1 NEW SIZE     NEW NAME   NEW SIZE
      PUT HENRY OUT "
      NEW LINE(HENRY OUT 2
      PUT HENRY OUT NEW NAME
      NEW LINE(HENRY OUT
    elsif TEMP TYPE DEFINE    FUNCTION TYPE then
      OUT PUT DATA(I TYPE OF      FUNCTION TYPE
      NEW SIZE               SIZE   9
      NEW NAME(1 9)           [illegible]
      NEW NAME(10 NEW SIZE      TEMP NAME    SIZE
      OUT PUT DATA I NAME OF 1 NEW SIZE     NEW NAME     [illegible]
      PUT HENRY OUT
      NEW LINE(HENRY OUT 2
      PUT HENRY OUT NEW NAME
```

```
  NEW_LINE(HENRY_OUT);
end if;

-- [illegible] a pretty name for the data file

TEMP1 := TEMP.NEXT1;
TEMP2 := TEMP1;

-- [illegible] variables increase global flow metric

LOOP
  EXIT WHEN (TEMP1.TYPE_DEFINE = END_FUNCTION_TYPE) OR
        (TEMP1.TYPE_DEFINE = END_PROCEDURE_CALL);
  if TEMP1.TYPE_DEFINE = ASSIGN_TYPE then
    ASSIGN_MARKER := TRUE;
  elsif TEMP1.TYPE_DEFINE = END_ASSIGN_TYPE then
    ASSIGN_MARKER := FALSE;
    GLOBAL_MARKER := FALSE;
  end if;
  if (TEMP1.IDENTITY = GLOBAL_DECLARE) AND (ASSIGN_MARKER) then
    [illegible]
    GLOBAL_READ := GLOBAL_READ + 1.0;
    if GLOBAL_MARKER then
      GLOBAL_READ_WRITE := GLOBAL_READ_WRITE + 1.0;
    end if;
    if (TEMP1.IDENTITY = GLOBAL_DECLARE) AND
      (NOT ASSIGN_MARKER) then
      GLOBAL_WRITE := GLOBAL_WRITE + 1.0;
      GLOBAL_MARKER := TRUE;
    end if;
  TEMP1 := TEMP2.NEXT1;
  TEMP2 := TEMP1;
end LOOP;

-- [illegible] of data flow procedure formal parameters

TEMP1 := TEMP.NEXT1;
  if TEMP1.TYPE_DEFINE = PARAM_TYPE then
    loop
    EXIT WHEN TEMP1.TYPE_DEFINE = END_PARAM_DECLARE;
    if TEMP1.PARAM_TYPE = IN_TYPE THEN
      FAN_IN := FAN_IN + 1.0;
      elsif TEMP1.PARAM_TYPE = OUT_TYPE THEN
      FAN_OUT := FAN_OUT + 1.0;
      elsif TEMP1.PARAM_TYPE = IN_OUT_TYPE THEN
      FAN_IN := FAN_IN + 1.0;
      FAN_OUT := FAN_OUT + 1.0;
    [illegible]
      TEMP2 := TEMP1;
      TEMP1 := TEMP2.NEXT1;
  end loop;
  [illegible]
```

```
--look for procedure and function type actual parameters

    TEMP     TEMP1.
    TEMP1    TEMP NEXT1
    TEMP     TEMP1.
    LOOP
      EXIT WHEN (TEMP TYPE DEFINE   END FUNCTION TYPE OR
            (TEMP TYPE DEFINE   END PROCEDURE CALL

      TEMP1    TEMP.
      TEMP     TEMP1 NEXT1
      if TEMP TYPE DEFINE   ASSIGN TYPE then
        ASSIGN MARKER    TRUE
      elsif TEMP TYPE DEFINE   END ASSIGN TYPE then
        ASSIGN MARKER    FALSE
        GLOBAL MARKER    FALSE
      end if
      if TEMP TYPE DEFINE   PROCEDURE TYPE then
        TEMP1    TEMP NEXT1
        LOOP
          EXIT WHEN TEMP1 TYPE DEFINE   END ACTUAL PARAM
          FAN OUT    FAN OUT   1.0
          TEMP2    TEMP1
          TEMP1    TEMP2 NEXT1
        END LOOP
      elsif TEMP TYPE DEFINE   FUNCTION TYPE THEN

-- count the function parameters

        TEMP1    TEMP NEXT1
        LOOP
          EXIT WHEN TEMP1 TYPE DEFINE   END ACTUAL PARAM
          FAN OUT    FAN OUT   1.0
          TEMP2    TEMP1
          TEMP1    TEMP2 NEXT1
        END LOOP
        FAN IN    FAN IN   1.0   RETURN FROM FUNCTION
      elsif (TEMP TYPE DEFINE   DATA STRUCTURE and
          (NOT ASSIGN MARK) THEN
        GLOBAL MARK    TRUE
        GLOBAL WRITE    GLOBAL WRITE   1
      elsif (TEMP TYPE DEFINE   DATA STRUCTURE [illegible]
          THEN
        if GLOBAL MARK THEN
          GLOBAL READ WRITE    GLOBAL READ WRITE [illegible]
        elsif NOT GLOBAL MARK THEN
          GLOBAL READ    GLOBAL READ   1
        end if
      end if
    END LOOP
```

```
-- now check for transitivity in the actual parameters

TOP    HEAD NEXT1
TEMP    TOP
LOOP
  EXIT WHEN TOP TYPE  DEFINE    END PACKAGE DECLARE;
  TOP     TEMP NEXT1
  TEMP    TOP
END LOOP
FOR I IN 1 PROC  FUNC  COUNT LOOP
  TEMP    HENRY  ARRAY(I) BEGIN POINTER
  PROC  PTR    TEMP NEXT1
  TEMP       PROC  PTR
LOOP
  LOOP
    EXIT WHEN (PROC  PTR TYPE  DEFINE    PROCEDURE TYPE) OR
           PROC  PTR TYPE  DEFINE    FUNCTION TYPE) OR
           PROC  PTR TYPE  DEFINE    END PROCEDURE CALL) OR
           PROC  PTR TYPE  DEFINE    END FUNCTION TYPE);
    PROC  PTR    TEMP NEXT1
    TEMP    PROC  PTR
  END LOOP
  IF  PROC  PTR TYPE  DEFINE     END PROCEDURE CALL) AND
     PROC  PTR TYPE  DEFINE     END FUNCTION  TYPE) THEN
    PARAM PTR    PROC  PTR NEXT1
    LOOP
      EXIT WHEN PARAM PTR TYPE  DEFINE    END  ACTUAL PARAM;
      NAME OF(I MAX     PARAM PTR NOMEN(I MAX)
      FAN IN    FAN IN    TRANSITIVITY IN(NAME OF
                         TOP PROC  PTR
      FAN OUT    FAN OUT    TRANSITIVITY OUT(NAME OF
                            PARAM PTR)
      TEMP    PARAM PTR
      PARAM PTR    TEMP NEXT1
    END LOOP
  END IF
  EXIT WHEN  PROC  PTR TYPE  DEFINE    END PROCEDURE CALL) OR
       [illegible] PTR TYPE  DEFINE    END FUNCTION TYPE) OR
       [illegible] PTR TYPE  DEFINE    END PACKAGE  TYPE);
  [illegible] PARAM PTR NEXT1
  TEMP [illegible] PTR
[illegible]
  [illegible]

[illegible] THE NUMBERS AND [illegible] IN OUTPUT FILE

[illegible] HENRY ARRAY [illegible] LENGTH POINTER
[illegible]
[illegible]
```

```
    GLOBAL_FLOW := GLOBAL_WRITE *
            (GLOBAL_READ - GLOBAL_READ_WRITE) -
            GLOBAL_READ_WRITE *
            (GLOBAL_READ - GLOBAL_READ_WRITE - 1.0).
    put(HENRY_OUT, "NUMBER OF LINES =    ");
    put(HENRY_OUT, LENGTH);
    OUT_PUT_DATA(I).CODE_LENGTH       :=  LENGTH;
    NEW_LINE(HENRY_OUT);
    put(HENRY_OUT, "FAN IN =          ");
    put(HENRY_OUT, FAN_IN);
    OUT_PUT_DATA(I).TYPE_FAN_IN       := FAN_IN;
    NEW_LINE(HENRY_OUT);
    put(HENRY_OUT, "FAN OUT =         ");
    put(HENRY_OUT, FAN_OUT);
    OUT_PUT_DATA(I).TYPE_FAN_OUT      := FAN_OUT;
    NEW_LINE(HENRY_OUT);
    put(HENRY_OUT, "COMPLEXITY =      ");
    put(HENRY_OUT, COMPLEXITY);
    OUT_PUT_DATA(I).TYPE_COMPLEXITY   := COMPLEXITY.
    NEW_LINE(HENRY_OUT);
    put(HENRY_OUT, "GLOBAL READ =     ");
    put(HENRY_OUT, GLOBAL_READ);
    OUT_PUT_DATA(I).TYPE_READ         := GLOBAL_READ.
    NEW_LINE(HENRY_OUT);
    put(HENRY_OUT, "GLOBAL WRITE =    ");
    put(HENRY_OUT, GLOBAL_WRITE);
    OUT_PUT_DATA(I).TYPE_WRITE        := GLOBAL_WRITE
    NEW_LINE(HENRY_OUT);
    put(HENRY_OUT, "GLOBAL READ WRITE = ");
    put(HENRY_OUT, GLOBAL_READ_WRITE);
    OUT_PUT_DATA(I).TYPE_READ_WRITE       GLOBAL_READ_WRITE.
    NEW_LINE(HENRY_OUT);
    put(HENRY_OUT, "GLOBAL FLOW       ").
    put(HENRY_OUT, GLOBAL_FLOW).
    OUT_PUT_DATA(I).TYPE_FLOW          GLOBAL_FLOW
    NEW_LINE(HENRY_OUT, 2);

  END LOOP;
  PUT(HENRY_OUT, "-----------------------------------")
 end if; --FIRST_HENRY_CALL.
 FIRST_HENRY_CALL := FALSE
 END CALCULATE_METRIC.
 END HENRY_ANALYSIS
```

```
MODULE NAME   : PACKAGE HENRY_DISPLAY
DATE CREATED  : [illegible] MAY 87
LAST MODIFIED : [illegible] MAY 87

AUTHOR        : [illegible] PAUL M. [illegible]

DESCRIPTION   : [illegible]

-------------------------------------------------------------------

[illegible]
```

```
procedure GRAPH_RELATIVE_DATA;
procedure PAUSE_PRINT(STOP_COUNT : in INTEGER;
            RUNNING_COUNT : in out INTEGER;
            DONE : in out BOOLEAN);
procedure SET_UP_SCREEN(IN_STRING : in ROW_STRING_TYPE;
            STRING_SIZE : in INTEGER);

procedure CENTER_STRING(NAME : in ROW_STRING_TYPE;
            IN_ROW_WIDTH : in integer);
end HENRY_DISPLAY;

package body HENRY_DISPLAY is

-- This procedure stops the printing of long lists so that the viewer
-- can pause to read without losing variable data. It also
-- [illegible] integer count

procedure PAUSE_PRINT(STOP_COUNT : in INTEGER;
            RUNNING_COUNT : in out INTEGER;
            DONE : in out BOOLEAN) is

  [illegible] CHARACTER [illegible]
  [illegible] ROW [illegible] INTEGER

  [illegible] RUNNING_COUNT then
  [illegible]
```

```
                    IN_ROW, WIDTH : in INTEGER) is

SCREEN_WIDTH : INTEGER := 76;
CENTER_POS : INTEGER := 0;
TEMP_NAME : ROW_STRING_TYPE;

begin
 FOR I IN 1..30 LOOP
  TEMP_NAME(I) := NULL_CHAR;
 END LOOP;
 TEMP_NAME(1..WIDTH) := NAME(1..WIDTH);
 CENTER_POS := SCREEN_WIDTH / 2 - WIDTH / 2;
 SET_CURSOR_POS(CENTER_POS, IN_ROW);
 PUT(TEMP_NAME);
 NEW_LINE;
end CENTER_STRING;

------------------------------------------------------------------------

--Puts the header of each data screen up with an underline to set it
--off from the data

procedure SET_UP_SCREEN(IN_STRING : in ROW_STRING_TYPE;
                    STRING_SIZE : in INTEGER) is

begin
 CLEARSCREEN;
 SET_REVERSE(ON);
 CENTER_STRING(IN_STRING, 1, STRING_SIZE);
 SET_REVERSE(OFF);
PUT("------------------------------------------------------------------");
 NEW_LINE(2);
END SET_UP_SCREEN;

------------------------------------------------------------------------

--lists the entire record stream of the Henry metric data

procedure LIST_HENRY_DATA is



SHORT_NAME_SIZE : constant integer := 1[illegible];
TEMP_POINTER_TI : POINTER := HEAD;
TEMP_NAME : LEXEME_TYPE;
SHORT_NAME : STRING(1..SHORT_NAME_SIZE);
SIZE : INTEGER;
[illegible] : INTEGER;
[illegible] : INTEGER := 2;
HEADER_STRING : ROW_STRING_TYPE;
[illegible] : BOOLEAN := FALSE;
```

```
HEADER_SIZE          : INTEGER := 21;

begin
  HEADER_STRING(1..HEADER_SIZE) := "LIST OF HENRY RECORDS";
  PUT(HENRY_FILE, "IN LIST_HENRY_DATA"); NEW_LINE(HENRY_FILE);
  LOOP EXIT WHEN DONE;
   SET_UP_SCREEN(HEADER_STRING, HEADER_SIZE);
   LOOP
    put("DECLARATION      ");
    case TEMP_POINTER.IDENTITY is
      when LOCAL_DECLARE        => put("Local declare");
      when GLOBAL_DECLARE       => put("Global declare");
      when others               => put("Undeclared");
    end case;
    new_line;
    put("NAME              ");
    if TEMP_POINTER.NOMEN(1) = NULL_CHAR then
      put("None");
    else
     TEMP_NAME(1..MAX_LINE_SIZE) :=
          TEMP_POINTER.NOMEN(1..MAX_LINE_SIZE);
     SHORT_NAME(1..SHORT_NAME_SIZE) := TEMP_NAME(1..SHORT_NAME_SIZE);
     put(SHORT_NAME);
     FOR I IN 1..SHORT_NAME_SIZE LOOP
      SHORT_NAME(I) := NULL_CHAR;
     END LOOP;
    [illegible]
    [illegible]
    put("ACTION TYPE       ");
    case TEMP_POINTER.TYPE_DEFINE is
    when UNDEFINED              => [illegible]
    when HENRY_HEADER_TYPE      => [illegible]
    [illegible]
```

```
[illegible] PARAMETER TYPE
[illegible] TEMP POINTER PARAM TYPE [illegible]
[illegible] IN TYPE [illegible]
[illegible] OUT TYPE [illegible]
[illegible] IN OUT TYPE [illegible]
[illegible]
[illegible]
[illegible]

[illegible]
[illegible] PRINT STOP RUNNING COUNT DONE
[illegible] RUNNING COUNT [illegible] AND NOT DONE THEN
RUNNING COUNT [illegible]
SET [illegible] SCREEN HEADER STRING HEADER SIZE

[illegible] TEMP POINTER NEXT [illegible]
TEMP POINTER [illegible]
EXIT WHEN TEMP POINTER [illegible] NULL OR DONE

[illegible] NOT DONE THEN
[illegible]
RUNNING COUNT [illegible]
[illegible]
[illegible]
[illegible] PRINT STOP RUNNING COUNT DONE

[illegible] HENRY DATA

[illegible]
[illegible]
[illegible]

[illegible] METRIC DATA [illegible]

[illegible] STRING [illegible]
[illegible] NATURAL
[illegible] POINTER [illegible] HEAD NEXT [illegible]
[illegible] INTEGER [illegible]
RUNNING COUNT INTEGER [illegible]
HEADER STRING ROW STRING TYPE
[illegible] BOOLEAN [illegible] FALSE
HEADER SIZE [illegible] INTEGER [illegible]

[illegible] HENRY FILE [illegible] METRIC DATA [illegible] NEW LINE HENRY FILE
HEADER STRING [illegible] HEADER SIZE [illegible] LIST HENRY METRIC VALUES
[illegible] OPEN HENRY OUT [illegible]
[illegible] FILE [illegible] MODE HENRY OUT [illegible]
```

```
    RESET(HENRY OUT, IN FILE);
  end if
else OPEN(HENRY OUT, IN FILE, HENRY_OUT_NAME);
end if
  SET UP SCREEN(HEADER STRING, HEADER_SIZE);
  IN STRING(1..8) := "PACKAGE ";
  IN STRING(9..49) := INPUT FILE NAME(1..41);
  put(IN STRING);
  NEW LINE(2);
  LOOP
    EXIT WHEN (END OF FILE(HENRY OUT) OR DONE);
    FOR J IN 1..49 LOOP
      IN STRING(J) := NULL CHAR;
    END LOOP;
    GET LINE(HENRY OUT, IN STRING, NUMBER_OF);
    PUT LINE(IN STRING);
    PAUSE PRINT(STOP RUNNING COUNT, DONE);
    IF (RUNNING COUNT = 0) AND (NOT DONE) THEN
      RUNNING COUNT := 1;
      SET UP SCREEN(HEADER STRING, HEADER SIZE);
    end if
  END LOOP
    IF (NOT DONE) THEN
      STOP := 1; RUNNING COUNT := 1;
      PAUSE PRINT(STOP, RUNNING COUNT, DONE);
    end if
  CLOSE(HENRY OUT);

end LIST METRIC DATA;
----------------------------------------------------------------

-- Lists the relative comparison metric data. This listing
-- compares each procedure/function analyzed with for example the
-- Henry numbers. It also gives a verbal report for each function/
-- procedure.

procedure WRITE RELATIVE DATA is

INDICATOR1,
INDICATOR2,
INDICATOR3     : FLOAT := 0.0;
UPPER LIMIT    : constant FLOAT := 4.0;
LOWER LIMIT    : constant FLOAT := 0.25;
TEMP HOLDER    : STRING(1..10);
STOP, RUNNING  : INTEGER := 1;
HEADER STRING  : ROW STRING TYPE;
ROW STRING     : ROW STRING TYPE;
SIZE           : INTEGER;
DONE           : BOOLEAN := FALSE;
HEADER SIZE    : INTEGER := 29;

begin
```

```
 HEADER_STRING(1..HEADER_SIZE) := "THE RELATIVE PERFORMANCE DATA";
 SET_UP_SCREEN(HEADER_STRING, HEADER_SIZE);
 if PROC_FUNC_COUNT < 16 THEN STOP := PROC_FUNC_COUNT;
 else STOP := 16;
 end if;
 PUT(HENRY_FILE, "IN WRITE RELATIVE DATA"); NEW_LINE(HENRY_FILE);

--name the outer loop so that can exit gracefully when the user
--wants to quit


OUTER_LOOP:
FOR J IN 1..7 LOOP
 CASE J is
  when 1 => ROW_STRING(1..6) := "FAN IN";
        SIZE := 6;
  when 2 => ROW_STRING(1..7) := "FAN OUT";
        SIZE := 7;
  when 3 => ROW_STRING(1..10) := "COMPLEXITY";
        SIZE := 10;
  when 4 => ROW_STRING(1..11) := "GLOBAL READ";
        SIZE := 11;
  when 5 => ROW_STRING(1..12) := "GLOBAL WRITE";
        SIZE := 12;
  when 6 => ROW_STRING(1..17) := "GLOBAL READ WRITE";
        SIZE := 17;
  when 7 => ROW_STRING(1..11) := "GLOBAL FLOW";
        SIZE := 11;
  when others => null;
  end case;
  CENTER_STRING(ROW_STRING, 4, SIZE);
  FOR I IN 1..PROC_FUNC_COUNT LOOP
   SET_CURSOR_POS(1, I - 5);
   REL_ARRAY(I).NAME_OF := OUT_PUT_DATA(I).NAME_OF;
   PUT(REL_ARRAY(I).NAME_OF); SET_CURSOR_POS(42, I - 5); PUT(" : ");

--set up the names before write the data

   CASE J is
    when 1 => put(REL_ARRAY(I).TYPE_FAN_IN);
    when 2 => put(REL_ARRAY(I).TYPE_FAN_OUT);
    when 3 => put(REL_ARRAY(I).TYPE_COMPLEXITY);
    when 4 => put(REL_ARRAY(I).TYPE_READ);
    when 5 => put(REL_ARRAY(I).TYPE_WRITE);
    when 6 => put(REL_ARRAY(I).TYPE_READ_WRITE);
    when 7 => put(REL_ARRAY(I).TYPE_FLOW);
    when others => null;
   end case;
   NEW_LINE;
   PAUSE_PRINT(STOP_RUNNING, DONE);

--boolean done is set true by user answering the query to quit
```

```
  EXIT OUTER_LOOP WHEN DONE;
  if (RUNNING = 0) AND (STOP = 16) THEN
     STOP := PROC_FUNC_COUNT - 17;
  elsif RUNNING = 0 THEN
     SET_UP_SCREEN(HEADER_STRING, HEADER_SIZE);
     RUNNING := 1;
   end if;
 end loop;
end loop OUTER_LOOP;

--set up to loop again once have cycled through to first stop
--count. This means have filled the screen once

STOP := 1; RUNNING := 1;
PAUSE_PRINT(STOP, RUNNING, DONE);

CLEARSCREEN;
PUT("The following are the maximums for each calculation: ");
new_line;
put("-------------------------------------------------");
new_line;
 put("Fan In            : "); put(REL_ARRAY(MAX_FAN_IN).NAME_OF); new_line;
 put("Fan Out           : "); put(REL_ARRAY(MAX_FAN_OUT).NAME_OF); new_line;
 put("Complexity        : "); put(REL_ARRAY(MAX_COMPLEXITY).NAME_OF); NEW_LINE;
 put("Global Read       : "); put(REL_ARRAY(MAX_READ).NAME_OF); NEW_LINE;
 PUT("Global Write      : "); put(REL_ARRAY(MAX_WRITE).NAME_OF); NEW_LINE;
 PUT("Global Read Write : "); put(REL_ARRAY(MAX_READ_WRITE).NAME_OF); NEW_LINE;
 PUT("Global Flow       : "); put(REL_ARRAY(MAX_FLOW).NAME_OF); NEW_LINE;
new_line;
put("-------------------------------------------------");
new_line;
STOP := 1; RUNNING := 1;
PAUSE_PRINT(STOP, RUNNING, DONE);
SET_UP_SCREEN(HEADER_STRING, HEADER_SIZE);

--calculate the indicator numbers so that can determine the relative
--performance of each procedure/function within each category

FOR I IN 1..PROC_FUNC_COUNT LOOP
 if REL_ARRAY(I).TYPE_FLOW /= 0.0 THEN
 INDICATOR1 := REL_ARRAY(I).TYPE_COMPLEXITY /
         REL_ARRAY(I).TYPE_FLOW;
 else INDICATOR1 := REL_ARRAY(I).TYPE_COMPLEXITY;
 end if;
 if REL_ARRAY(I).TYPE_FAN_OUT /= 0.0 THEN
 INDICATOR2 := REL_ARRAY(I).TYPE_FAN_IN /
         REL_ARRAY(I).TYPE_FAN_OUT;
 else INDICATOR2 := REL_ARRAY(I).TYPE_FAN_IN;
 end if;
 if REL_ARRAY(I).TYPE_WRITE /= 0.0 THEN
 INDICATOR3 := REL_ARRAY(I).TYPE_READ /
         REL_ARRAY(I).TYPE_WRITE;
```

```
 else INDICATOR3    REL ARRAY(I) TYPE READ
 end if
PUT(REL ARRAY(I).NAME OF); put("  ");
new line;

--put out the results of the indicator analysis

 IF INDICATOR1    UPPER LIMIT THEN
   PUT(" - Has significant complexity compared to global data flow ")
   new line;
   if INDICATOR2  UPPER LIMIT THEN
     put(" - This implies poor internal code structure. ");
     new line;
     put(" - Consider remodularization ");
     new line;
   elsif INDICATOR2    LOWER LIMIT THEN
     PUT(" - This implies an extremely complex interface ")
     new line;
   end if
 ELSIF INDICATOR1    LOWER LIMIT THEN
  PUT(" - Has significant global data flow compared to complexity ");
  new line;
  if INDICATOR3    UPPER LIMIT THEN
    put(" - This implies an overworked data structure  ")
    new line;
    put(" - or a considerable number of function calls ");
    new line
    put(" - Consider redistributing the data structure into this module")
    new line;
  elsif INDICATOR3    LOWER LIMIT THEN
    PUT(" - This implies a program stress point");
    new line;
    put(" - or a critical data flow point");
    new line
    put(" - Consider reorganizing the data structure.");
    new line
  end if;
 ELSE
    TEMP HOLDER(1 10)    REL ARRAY(I).NAME OF(1 10);
    put(" - Is a fairly well balanced "); put(TEMP HOLDER).
    new line
    put(" - This implies good modularization. ");
    new line
 END IF;
STOP    1; RUNNING    1;
PAUSE PRINT(STOP, RUNNING, DONE);
EXIT WHEN DONE
end loop;
 if NOT DONE THEN
STOP    1; RUNNING    1;
PAUSE PRINT(STOP, RUNNING, DONE);
end if
```

```
end WRITE_RELATIVE_DATA;


--[illegible]

procedure GRAPH_RELATIVE_DATA is

LOOP_CNT       : INTEGER;
ROW_STRING     : ROW_STRING_TYPE;
HEADER_STRING  : ROW_STRING_TYPE;
SIZE           : INTEGER;
STOP_RUNNING   : INTEGER := 1;
DONE           : BOOLEAN := FALSE;
SCALE          : INTEGER := 5;
NUM_LOOP_CNT,
REM_CNT        : INTEGER;
HEADER_SIZE    : INTEGER := 30;

begin
 NUM_LOOP_CNT := PROC_FUNC_COUNT / 5;
 REM_CNT := PROC_FUNC_COUNT REM 5;

--loop count is the number of screens need to display
--remainder count is the partial screen that is left over

 HEADER_STRING(1..30) := "THE GRAPHICAL PERFORMANCE DATA";
 if NUM_LOOP_CNT >= 1 THEN STOP := 5;
 else STOP := REM_CNT;
 end if;
 PUT(HENRY_FILE,"IN WRITE RELATIVE DATA"); NEW_LINE(HENRY_FILE);
 SET_UP_SCREEN(HEADER_STRING, HEADER_SIZE);

--set up to exit gracefully when the user wants to quit


GRAPH_LOOP:
FOR J IN 1..7 LOOP
 CASE J is
  when 1 => ROW_STRING(1..6) := "FAN IN";
        SIZE := 6;
  when 2 => ROW_STRING(1..7) := "FAN OUT";
        SIZE := 7;
  when 3 => ROW_STRING(1..10) := "COMPLEXITY";
        SIZE := 10;
  when 4 => ROW_STRING(1..11) := "GLOBAL READ";
        SIZE := 11;
  when 5 => ROW_STRING(1..12) := "GLOBAL WRITE";
        SIZE := 12;
  when 6 => ROW_STRING(1..17) := "GLOBAL READ WRITE";
        SIZE := 17;
  when 7 => ROW_STRING(1..11) := "GLOBAL FLOW";
        SIZE := 11;
```

```
[illegible]

[illegible] HEADER_STRING, HEADER_SIZE [illegible]
[illegible] LOOP_CNT [illegible]

[illegible] ARRAY [illegible] NAME [illegible] DATA [illegible] NAME [illegible]
[illegible] ARRAY [illegible] NAME [illegible]
CASE [illegible]
[illegible] LOOP_CNT [illegible] SCALE * INTEGER [illegible] ARRAY [illegible] TYPE [illegible]
[illegible] LOOP_CNT [illegible] SCALE * INTEGER [illegible] ARRAY [illegible] TYPE [illegible]
[illegible] LOOP_CNT [illegible] SCALE * INTEGER [illegible] ARRAY [illegible] TYPE [illegible]
[illegible] LOOP_CNT [illegible] SCALE * INTEGER [illegible] ARRAY [illegible] TYPE READ
[illegible] LOOP_CNT [illegible] SCALE * INTEGER [illegible] ARRAY [illegible] TYPE WRITE
[illegible] LOOP_CNT [illegible] SCALE * INTEGER [illegible] ARRAY [illegible] TYPE READ_WRITE
[illegible] LOOP_CNT [illegible] SCALE * INTEGER [illegible] ARRAY [illegible] TYPE [illegible]
[illegible]
[illegible]

[illegible]
[illegible]

FOR I IN 1..LOOP_CNT LOOP
 PUT("*");
END LOOP;
NEW_LINE(1);
PAUSE_PRINT(STOP, RUNNING, DONE);
EXIT GRAPH_LOOP WHEN DONE;

-- [illegible] actually prints the bar chart

if (RUNNING = 0) AND (STOP /= 5) THEN
 if PROC_FUNC_COUNT < 5 THEN
   STOP := 5;
 end if;
   SET_UP_SCREEN(HEADER_STRING, HEADER_SIZE);
   RUNNING := 1;
elsif (RUNNING = 0) AND (STOP = 5) THEN
 if PROC_FUNC_COUNT < 5 THEN
   SET_UP_SCREEN(HEADER_STRING, HEADER_SIZE);
   RUNNING := 1;
  elsif PROC_FUNC_COUNT > 5 THEN
   NUM_LOOP_CNT := NUM_LOOP_CNT - 1;
   if NUM_LOOP_CNT = 1 THEN STOP := 5; RUNNING := 1;
   elsif REM_CNT /= 0 THEN
      STOP := REM_CNT; RUNNING := 1;
   else SET_UP_SCREEN(HEADER_STRING, HEADER_SIZE);
      RUNNING := 1;
   end if;
  end if;
```

```
[illegible]

[illegible]

      EXECUTE [illegible]

   CALCULATE RELATIVE DATA

[illegible]

   VIEW HENRY SELECTOR [illegible]

NORMALIZER [illegible] FLOAT [illegible] 1

begin
  PUT HENRY FILE  IN VIEW HENRY PROCE  URE   NEW  LINE(HENRY FILE
  CALCULATE  METRIC(HEAD  HEAD LINE

  -- find the maximums for each variable

  FOR I IN 1 PROC  FUNC  COUNT LOOP
   IF OUT  PUT  DATA(I) TYPE  FAN IN
     OUT  PUT  DATA(MAX  FAN  IN  TYPE  FAN  IN THEN
      MAX  FAN  IN     I
   END IF
   IF OUT  PUT  DATA(I) TYPE  FAN  OUT
     OUT  PUT  DATA(MAX  FAN  OUT) TYPE  FAN  OUT THEN
     MAX  FAN  OUT     I
   END IF
   IF OUT  PUT  DATA(I) TYPE  COMPLEXITY
     OUT  PUT  DATA(MAX  COMPLEXITY) TYPE  COMPLEXITY THEN
      MAX  COMPLEXITY     I
   END IF
   IF OUT  PUT  DATA(I) TYPE  READ
     OUT  PUT  DATA(MAX  READ) TYPE  READ THEN
      MAX  READ     I
   END IF
   IF OUT  PUT  DATA(I) TYPE  WRITE
     OUT  PUT  DATA(MAX  WRITE) TYPE  WRITE THEN
      MAX  WRITE     I
   END IF
```

```
if OUT PUT DATA(MAX FAN IN) TYPE FAN IN ... THEN
REL ARRAY(I) TYPE FAN IN = NORMALIZER *
(OUT PUT DATA(I) TYPE FAN IN /
            OUT PUT DATA(MAX FAN IN) TYPE FAN IN)
else REL ARRAY(I) TYPE FAN IN

if OUT PUT DATA(MAX FAN OUT) TYPE FAN OUT          THEN
REL ARRAY(I) TYPE FAN OUT = NORMALIZER *
(OUT PUT DATA(I) TYPE FAN OUT /
            OUT PUT DATA(MAX FAN OUT) TYPE FAN OUT)
else REL ARRAY(I) TYPE FAN OUT
end if

if OUT PUT DATA(MAX COMPLEXITY) TYPE COMPLEXITY          THEN
REL ARRAY(I) TYPE COMPLEXITY = NORMALIZER *
(OUT PUT DATA(I) TYPE COMPLEXITY /
            OUT PUT DATA(MAX COMPLEXITY) TYPE COMPLEXITY)
else REL ARRAY(I) TYPE COMPLEXITY = 0.0
end if

if OUT PUT DATA(MAX READ) TYPE READ     0.0 THEN
REL ARRAY(I) TYPE READ = NORMALIZER *
(OUT PUT DATA(I) TYPE READ /
            OUT PUT DATA(MAX READ) TYPE READ)
else REL ARRAY(I) TYPE READ = 0.0
end if

if OUT PUT DATA(MAX WRITE) TYPE WRITE     0.0 THEN
REL ARRAY(I) TYPE WRITE = NORMALIZER *
(OUT PUT DATA(I) TYPE WRITE /
            OUT PUT DATA(MAX WRITE) TYPE WRITE)
else REL ARRAY(I) TYPE WRITE = 0.0
end if

if OUT PUT DATA(MAX READ WRITE) TYPE READ WRITE     0.0 THEN
REL ARRAY(I) TYPE READ WRITE = NORMALIZER *
(OUT PUT DATA(I) TYPE READ WRITE
```

APPENDIX C. MODIFIED PARSERS

[illegible]

package BYPASS FUNCTION [illegible]

[illegible]

```
package body BYPASS FUNCTION is

    -- This function compares the lexeme of the current token with the
    -- token currently being sought by the parser. If the current token
    -- type is identifier then a test is conducted to ensure it is not
    -- a reserved word
function BYPASS(TOKEN ARRAY ENTRY CODE : integer) return boolean is
  CONSUME              : boolean := FALSE;
  LEXEME               : string(1..LINESIZE);
  SIZE                 : natural;
  HENRY LEXEME         : string(1..MAX LINE SIZE);

begin
  GET CURRENT TOKEN RECORD(CURRENT TOKEN RECORD LEXEME LENGTH);
  LEXEME := CURRENT TOKEN RECORD.LEXEME;
  SIZE := CURRENT TOKEN RECORD.LEXEME SIZE - 1;
```

```
[illegible]

    [illegible] UPPER CASE HENRY LEXEME [illegible]
    [illegible] UPPER CASE LEXEME SIZE
    HENRY WRITE ENABLE [illegible]
  [illegible]
    WRITE HENRY DATA(BLANK HENRY LEXEME IDENT TYPE [illegible]
    CREATE NODE(NEXT HEN LAST RECORD)
    HENRY WRITE ENABLE    FALSE
   end if
   OPERAND METRIC(HEAD NODE CURRENT TOKEN RECORD [illegible]
   DECLARE TYPE    VARIABLE DECLARE
  end if

when TOKEN NUMERIC LITERAL
  if (CURRENT TOKEN RECORD TOKEN TYPE    NUMERIC LIT) then
   CONSUME    TRUE
   DECLARE TYPE    CONSTANT DECLARE
   OPERAND METRIC(HEAD NODE CURRENT TOKEN RECORD DECLARE TYPE)
   DECLARE TYPE    VARIABLE DECLARE
   if HENRY WRITE ENABLE then
     WRITE HENRY DATA(LOCAL DECLARE HENRY LEXEME IDENT TYPE
             NONE NEXT HEN)
     CREATE NODE(NEXT HEN LAST RECORD)
     HENRY WRITE ENABLE    FALSE
   end if
  end if

when TOKEN CHARACTER LITERAL
  if (CURRENT TOKEN RECORD TOKEN TYPE    CHARACTER LIT) then
   if HENRY WRITE ENABLE then
     WRITE HENRY DATA(LOCAL DECLARE HENRY LEXEME IDENT TYPE
             NONE NEXT HEN)
     CREATE NODE(NEXT HEN LAST RECORD)
     HENRY WRITE ENABLE    FALSE
   end if
   CONSUME    TRUE
  end if

when TOKEN STRING LITERAL
  if (CURRENT TOKEN RECORD TOKEN TYPE    STRING LIT) then
```

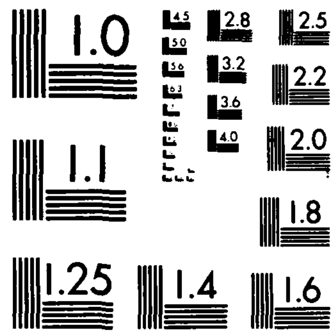
```
[illegible]

[illegible]

when TOKEN_FOR =>
  if (ADJUST_LEXEME(LEXEME_SIZE) = "for") then
    CONSUME := TRUE;
  end if;

when TOKEN_OTHERS =>
  if (ADJUST_LEXEME(LEXEME_SIZE) = "others") then
    CONSUME := TRUE;
  end if;

when TOKEN_RETURN =>
  if (ADJUST_LEXEME(LEXEME_SIZE) = "return") then
    CONSUME := TRUE;
  end if;

when TOKEN_EXIT =>
  if (ADJUST_LEXEME(LEXEME_SIZE) = "exit") then
    CONSUME := TRUE;
  end if;

when TOKEN_PROCEDURE =>
  if (ADJUST_LEXEME(LEXEME_SIZE) = "procedure") then
    CONSUME := TRUE;
  end if;

when TOKEN_FUNCTION =>
  if (ADJUST_LEXEME(LEXEME_SIZE) = "function") then
    CONSUME := TRUE;
  end if;

when TOKEN_WITH =>
  if (ADJUST_LEXEME(LEXEME_SIZE) = "with") then
    CONSUME := TRUE;
  end if;

when TOKEN_USE =>
  if (ADJUST_LEXEME(LEXEME_SIZE) = "use") then
    CONSUME := TRUE;
  end if;
```

```
when TOKEN_PACKAGE =>
  if (ADJUST_LEXEME(LEXEME_SIZE) = "package") then
    CONSUME := TRUE;
  end if;

when TOKEN_BODY =>
  if (ADJUST_LEXEME(LEXEME_SIZE) = "body") then
    CONSUME := TRUE;
  end if;

when TOKEN_RANGE =>
  if (ADJUST_LEXEME(LEXEME_SIZE) = "range") then
    CONSUME := TRUE;
  end if;

when TOKEN_IN =>
  if (ADJUST_LEXEME(LEXEME_SIZE) = "in") then
    CONSUME := TRUE;
  end if;

when TOKEN_OUT =>
  if (ADJUST_LEXEME(LEXEME_SIZE) = "out") then
    CONSUME := TRUE;
  end if;

when TOKEN_SUBTYPE =>
  if (ADJUST_LEXEME(LEXEME_SIZE) = "subtype") then
    CONSUME := TRUE;
  end if;

when TOKEN_TYPE =>
  if (ADJUST_LEXEME(LEXEME_SIZE) = "type") then
    CONSUME := TRUE;
  end if;

when TOKEN_IS =>
  if (ADJUST_LEXEME(LEXEME_SIZE) = "is") then
    CONSUME := TRUE;
  end if;

when TOKEN_NULL =>
  if (ADJUST_LEXEME(LEXEME_SIZE) [illegible]
    CONSUME := TRUE;
  end if;

when TOKEN_ACCESS
  if (ADJUST_LEXEME(LEXEME_SIZE [illegible]
    CONSUME := TRUE
  end if

when TOKEN_ARRAY
  if (ADJUST_LEXEME(LEXEME_SIZE [illegible]
```

```
[illegible]NSUME [illegible]TRUE;
[illegible]

[illegible]KEN_[illegible]S =>
[illegible] LEXEME(LEXEME_SIZE) [illegible]s") then
[illegible]NSUME [illegible]TRUE;
[illegible]

[illegible]KEN_[illegible]TA =>
[illegible] LEXEME(LEXEME_SIZE) [illegible]ta") then
[illegible]NSUME [illegible]
[illegible]

[illegible]KEN_[illegible] =>
[illegible] LEXEME(LEXEME_SIZE) [illegible]ord") then
[illegible]

[illegible]KEN_[illegible]ANT =>
[illegible] LEXEME(LEXEME_SIZE) [illegible]stant") then
[illegible]

[illegible]KEN_[illegible]W =>
[illegible] LEXEME(LEXEME_SIZE) [illegible]ew") then
[illegible]

[illegible]KEN_[illegible]ION =>
[illegible] LEXEME(LEXEME_SIZE) "exception") then
[illegible]TRUE;

[illegible]KEN_[illegible]NAMES =>
[illegible] LEXEME(LEXEME_SIZE) "renames") then
[illegible]TRUE;

[illegible]KEN_[illegible]IVATE =>
[illegible] LEXEME(LEXEME_SIZE) "private") then
[illegible]TRUE;

[illegible]KEN_[illegible]MITED =>
[illegible] LEXEME(LEXEME_SIZE) "limited") then
[illegible]TRUE;

[illegible]KEN_TASK =>
[illegible]MATCHES_LEXEME(LEXEME_SIZE) "task") then
[illegible]NSUME := TRUE;
[illegible]
```

AD-A184 805 ADAMEASURE: AN IMPLEMENTATION OF THE HALSTEAD AND HENRY METRICS(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA P M HERZIG JUN 87 2/2

UNCLASSIFIED F/G 12/5 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

```
when TOKEN_ENTRY =>
   if (ADJUST_LEXEME(LEXEME, SIZE) = "entry") then
     CONSUME := TRUE;
   end if;

when TOKEN_ACCEPT =>
   if (ADJUST_LEXEME(LEXEME, SIZE) = "accept") then
     CONSUME := TRUE;
   end if;

when TOKEN_DELAY =>
   if (ADJUST_LEXEME(LEXEME, SIZE) = "delay") then
     CONSUME := TRUE;
   end if;

when TOKEN_SELECT =>
   if (ADJUST_LEXEME(LEXEME, SIZE) = "select") then
     CONSUME := TRUE;
   end if;

when TOKEN_TERMINATE =>
   if (ADJUST_LEXEME(LEXEME, SIZE) = "terminate") then
     CONSUME := TRUE;
   end if;

when TOKEN_ABORT =>
   if (ADJUST_LEXEME(LEXEME, SIZE) = "abort") then
     CONSUME := TRUE;
   end if;

when TOKEN_SEPARATE =>
   if (ADJUST_LEXEME(LEXEME, SIZE) = "separate") then
     CONSUME := TRUE;
   end if;

when TOKEN_RAISE =>
   if (ADJUST_LEXEME(LEXEME, SIZE) = "raise") then
     CONSUME := TRUE;
   end if;

when TOKEN_GENERIC =>
   if (ADJUST_LEXEME(LEXEME, SIZE) = "generic") then
     CONSUME := TRUE;
   end if;

when TOKEN_AT =>
   if (ADJUST_LEXEME(LEXEME, SIZE) = "at") then
     CONSUME := TRUE;
   end if;

when TOKEN_REVERSE =>
```

```
    if (ADJUST_LEXEME(LEXEME, SIZE) = "reverse") then
      CONSUME := TRUE;
    end if;

  when TOKEN_DO =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "do") then
      CONSUME := TRUE;
    end if;

  when TOKEN_GOTO =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "goto") then
      CONSUME := TRUE;
    end if;

  when TOKEN_OF =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "of") then
      CONSUME := TRUE;
    end if;

  when TOKEN_ALL =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "all") then
      CONSUME := TRUE;
    end if;

  when TOKEN_PRAGMA =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "pragma") then
      CONSUME := TRUE;
    end if;

  when TOKEN_AND =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "and") then
      CONSUME := TRUE;
    end if;
    OPERATOR_METRIC(TOKEN_AND, CONSUME, RESERVE_WORD_TEST);

  when TOKEN_OR =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "or") then
      CONSUME := TRUE;
    end if;
    OPERATOR_METRIC(TOKEN_OR, CONSUME, RESERVE_WORD_TEST);

  when TOKEN_NOT =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "not") then
      CONSUME := TRUE;
    end if;
    OPERATOR_METRIC(TOKEN_NOT, CONSUME, RESERVE_WORD_TEST);

  when TOKEN_XOR =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "xor") then
      CONSUME := TRUE;
    end if;
    OPERATOR_METRIC(TOKEN_XOR, CONSUME, RESERVE_WORD_TEST);
```

```
when TOKEN_MOD =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "mod") then
    CONSUME := TRUE;
  end if;
  OPERATOR_METRIC(TOKEN_MOD, CONSUME, RESERVE_WORD_TEST);

when TOKEN_REM =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "rem") then
    CONSUME := TRUE;
  end if;
  OPERATOR_METRIC(TOKEN_REM, CONSUME, RESERVE_WORD_TEST);

when TOKEN_ABSOLUTE =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "abs") then
    CONSUME := TRUE;
  end if;
  OPERATOR_METRIC(TOKEN_ABSOLUTE, CONSUME, RESERVE_WORD_TEST);

when TOKEN_ASTERISK =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "*") then
    CONSUME := TRUE;
  end if;
  OPERATOR_METRIC(TOKEN_ASTERISK, CONSUME, RESERVE_WORD_TEST);

when TOKEN_SLASH =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "/") then
    CONSUME := TRUE;
  end if;
  OPERATOR_METRIC(TOKEN_SLASH, CONSUME, RESERVE_WORD_TEST);

when TOKEN_EXPONENT =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "**") then
    CONSUME := TRUE;
  end if;
  OPERATOR_METRIC(TOKEN_EXPONENT, CONSUME, RESERVE_WORD_TEST);

when TOKEN_PLUS =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "+") then
    CONSUME := TRUE;
  end if;
  OPERATOR_METRIC(TOKEN_PLUS, CONSUME, RESERVE_WORD_TEST);

when TOKEN_MINUS =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "-") then
    CONSUME := TRUE;
  end if;
  OPERATOR_METRIC(TOKEN_MINUS, CONSUME, RESERVE_WORD_TEST);

when TOKEN_AMPERSAND =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "&") then
    CONSUME := TRUE;
```

```
    end if;
    OPERATOR_METRIC(TOKEN_AMPERSAND, CONSUME, RESERVE_WORD_TEST);

  when TOKEN_EQUALS =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "=") then
      CONSUME := TRUE;
    end if;
    OPERATOR_METRIC(TOKEN_EQUALS, CONSUME, RESERVE_WORD_TEST);

  when TOKEN_NOT_EQUALS =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "/=") then
      CONSUME := TRUE;
    end if;
    OPERATOR_METRIC(TOKEN_NOT_EQUALS, CONSUME, RESERVE_WORD_TEST);

  when TOKEN_LESS_THAN =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "<") then
      CONSUME := TRUE;
    end if;
    OPERATOR_METRIC(TOKEN_LESS_THAN, CONSUME, RESERVE_WORD_TEST);

  when TOKEN_LESS_THAN_EQUALS =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = "<=") then
      CONSUME := TRUE;
    end if;
    OPERATOR_METRIC(TOKEN_LESS_THAN_EQUALS, CONSUME, RESERVE_WORD_TEST

  when TOKEN_GREATER_THAN =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = ">") then
      CONSUME := TRUE;
    end if;
    OPERATOR_METRIC(TOKEN_GREATER_THAN, CONSUME, RESERVE_WORD_TEST);

  when TOKEN_GREATER_THAN_EQUALS =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = ">=") then
    CONSUME := TRUE;
    end if;
    OPERATOR_METRIC(TOKEN_GREATER_THAN_EQUALS, CONSUME, RESERVE_WORD_T

  when TOKEN_ASSIGNMENT =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = ":=") then
      CONSUME := TRUE;
      OPERATOR_METRIC(TOKEN_ASSIGNMENT, CONSUME, RESERVE_WORD_TEST);
    end if;

  when TOKEN_COMMA =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = ",") then
      CONSUME := TRUE;
    end if;

  when TOKEN_SEMICOLON =>
    if (ADJUST_LEXEME(LEXEME, SIZE) = ";") then
```

```
    UPDATE_LINE_COUNT;
    CONSUME := TRUE;
  end if;

when TOKEN_PERIOD =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = ".") then
    CONSUME := TRUE;
  end if;

when TOKEN_LEFT_PAREN =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "(") then
    CONSUME := TRUE;
  end if;

when TOKEN_RIGHT_PAREN =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = ")") then
    CONSUME := TRUE;
  end if;

when TOKEN_COLON =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = ":") then
    CONSUME := TRUE;
  end if;

when TOKEN_APOSTROPHE =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "'") then
    CONSUME := TRUE;
  end if;

when TOKEN_RANGE_DOTS =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "..") then
    CONSUME := TRUE;
  end if;

when TOKEN_ARROW =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "=>") then
    CONSUME := TRUE;
  end if;

when TOKEN_BAR =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "|") then
    CONSUME := TRUE;
  end if;

when TOKEN_BRACKETS =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "<>") then
    CONSUME := TRUE;
  end if;

when TOKEN_LEFT_BRACKET =>
  if (ADJUST_LEXEME(LEXEME, SIZE) = "<<") then
    CONSUME := TRUE;
```

```
      end if;

    when TOKEN_RIGHT_BRACKET =>
      if (ADJUST_LEXEME(LEXEME, SIZE) = ">>") then
        CONSUME := TRUE;
      end if;

    when others => null;
  end case;

  ADJUST_TOKEN_BUFFER(CONSUME, RESERVE_WORD_TEST);

  return (CONSUME);
end BYPASS;

---------------------------------------------------------------------------

  -- this procedure tests all identifiers to verify they are not reserved
  -- words. The most common reserved words are tested first and the process
  -- halts when a match is made or the test fails.
procedure CONDUCT_RESERVE_WORD_TEST(CONSUME : in out boolean) is
begin
  RESERVE_WORD_TEST := TRUE;
  for RESERVE_WORD_INDEX in TOKEN_END..TOKEN_ABSOLUTE loop
    if (BYPASS(RESERVE_WORD_INDEX)) then
      CONSUME := FALSE;
    end if;
    exit when not CONSUME;
  end loop;
  RESERVE_WORD_TEST := FALSE;
end CONDUCT_RESERVE_WORD_TEST;

end BYPASS_FUNCTION;


--*****************************************************************--
--
-- TITLE:         AN ADA SOFTWARE METRIC
--
-- MODULE NAME:   PACKAGE PARSER_0
-- DATE CREATED:  09 OCT 86
-- LAST MODIFIED: 30 MAY 87
--
-- AUTHORS:       LCDR JEFFREY L. NIEDER
--                LT KARL S. FAIRBANKS, JR.
--                LCDR PAUL M. HERZIG
-- DESCRIPTION: This package contains eight functions that
--     make up the highest level productions for our top-down,
--     recursive descent parser.
--
--*****************************************************************--
```

```
with PARSER_1, PARSER_2, PARSER_3, HENRY_GLOBAL, HENRY, BYPASS_FUNCTION,
   HALSTEAD_METRIC, GLOBAL_PARSER, GLOBAL, TEXT_IO;
use PARSER_1, PARSER_2, PARSER_3, HENRY_GLOBAL, HENRY, BYPASS_FUNCTION,
   HALSTEAD_METRIC, GLOBAL_PARSER, GLOBAL, TEXT_IO;

package PARSER_0 is
 function COMPILATION return boolean;
 function COMPILATION_UNIT return boolean;
 function CONTEXT_CLAUSE return boolean;
 function BASIC_UNIT return boolean;
 function LIBRARY_UNIT return boolean;
 function SECONDARY_UNIT return boolean;
 function LIBRARY_UNIT_BODY return boolean;
 function SUBUNIT return boolean;
end PARSER_0;

----------------------------------------------------------------------------
----------------------------------------------------------------------------

package body PARSER_0 is

 -- COMPILATION --> {COMPILATION_UNIT}+
function COMPILATION return boolean is
begin
put("In compilation "); new_line;
put(RESULT_FILE, "In compilation "); new_line(RESULT_FILE);
 if (COMPILATION_UNIT) then
   while (COMPILATION_UNIT) loop
     null;
   end loop;
   return (TRUE);
 else
   return (FALSE);
 end if;
end COMPILATION;

----------------------------------------------------------------------------

 -- COMPILATION_UNIT --> CONTEXT_CLAUSE BASIC_UNIT
function COMPILATION_UNIT return boolean is
begin
put(RESULT_FILE, "In compilation_unit "); new_line(RESULT_FILE);
 if (CONTEXT_CLAUSE) then
   if (BASIC_UNIT) then
     return (TRUE);
   else
     return (FALSE);
   end if;
 else
   return (FALSE);
 end if;
end COMPILATION_UNIT;
```

```
---------------------------------------------------------------------------

 -- CONTEXT_CLAUSE --> {with WITH_OR_USE_CLAUSE {use WITH_OR_USE_CLAUSE}* }*
function CONTEXT_CLAUSE return boolean is
begin
put(RESULT_FILE, "In context_clause "); new_line(RESULT_FILE);
 while (BYPASS(TOKEN_WITH)) loop
   if not (WITH_OR_USE_CLAUSE) then
     SYNTAX_ERROR("Context clause");
   end if;
   while (BYPASS(TOKEN_USE)) loop
     if not (WITH_OR_USE_CLAUSE) then
       SYNTAX_ERROR("Context clause");
     end if;
   end loop;                    -- inner while loop
 end loop;                             -- outer while loop
 return (TRUE);
end CONTEXT_CLAUSE;

---------------------------------------------------------------------------

 -- BASIC_UNIT --> LIBRARY_UNIT
 --            --> SECONDARY_UNIT
function BASIC_UNIT return boolean is
begin
put(RESULT_FILE, "In basic_unit "); new_line(RESULT_FILE);
 if (LIBRARY_UNIT) then
   return (TRUE);
 elsif (SECONDARY_UNIT) then
   return (TRUE);
 else
   return (FALSE);
 end if;
end BASIC_UNIT;

---------------------------------------------------------------------------

 -- LIBRARY_UNIT --> procedure PROCEDURE_UNIT
 --              --> function FUNCTION_UNIT
 --              --> package PACKAGE_DECLARATION
 --              --> generic GENERIC_DECLARATION
function LIBRARY_UNIT return boolean is
begin
put(RESULT_FILE. "In library_unit "); new_line(RESULT_FILE);
 if (BYPASS(TOKEN_PROCEDURE)) then
   DECLARE_TYPE := PROCEDURE_DECLARE;
   if (PROCEDURE_UNIT) then
     return (TRUE);
   else
     SYNTAX_ERROR("Library unit");
   end if;                            -- if procedure_unit statement
 elsif (BYPASS(TOKEN_FUNCTION)) then
```

```
  DECLARE_TYPE := FUNCTION_DECLARE;
  if (FUNCTION_UNIT) then
   return (TRUE);
  else
   SYNTAX_ERROR("Library unit");
  end if;                              -- if function_unit statement
 elsif (BYPASS(TOKEN_PACKAGE)) then
  DECLARE_TYPE := PACKAGE_DECLARE;
  if (PACKAGE_DECLARATION) then
   return (TRUE);
  else
   SYNTAX_ERROR("Library unit");
  end if;                              -- if package_declaration
 elsif (BYPASS(TOKEN_GENERIC)) then
  if (GENERIC_DECLARATION) then
   return (TRUE);
  else
   SYNTAX_ERROR("Library unit");
  end if;                              -- if generic_declaration
 else
  return (FALSE);
 end if;
end LIBRARY_UNIT;

-----------------------------------------------------------------------------

 -- SECONDARY_UNIT --> LIBRARY_UNIT_BODY
 --                --> SUBUNIT
function SECONDARY_UNIT return boolean is
begin
put(RESULT_FILE, "In secondary_unit "); new_line(RESULT_FILE);
 if (LIBRARY_UNIT_BODY) then
  return (TRUE);
 elsif (SUBUNIT) then
  return (TRUE);
 else
  return (FALSE);
 end if;
end SECONDARY_UNIT;

-----------------------------------------------------------------------------

 -- LIBRARY_UNIT_BODY --> procedure PROCEDURE_UNIT
 --                   --> function FUNCTION_UNIT
 --                   --> package PACKAGE_DECLARATION
 --                   --> generic GENERIC_DECLARATION
function LIBRARY_UNIT_BODY return boolean is
begin
put(RESULT_FILE, "In library_unit_body "); new_line(RESULT_FILE);
 if (BYPASS(TOKEN_PROCEDURE)) then
  DECLARE_TYPE := PROCEDURE_DECLARE;
  if (PROCEDURE_UNIT) then
```

```
      return (TRUE);
    else
    SYNTAX_ERROR("Library unit body");
    end if;                               -- if procedure_unit statement
  elsif (BYPASS(TOKEN_FUNCTION)) then
    DECLARE_TYPE := FUNCTION_DECLARE;
    if (FUNCTION_UNIT) then
      return (TRUE);
    else
      SYNTAX_ERROR("Library unit body");
    end if;                               -- if function_unit statement
  elsif (BYPASS(TOKEN_PACKAGE)) then
    DECLARE_TYPE := PACKAGE_DECLARE;
    HENRY_WRITE_ENABLE := TRUE;
    put(result_file, "true"); new_line(result_file);
    if (PACKAGE_DECLARATION) then
      return (TRUE);
    else
      SYNTAX_ERROR("Library unit body");
    end if;                               -- if package_declaration
  else
    return (FALSE);
  end if;                   -- if bypass(token_procedure)
end LIBRARY_UNIT_BODY;

-----------------------------------------------------------------------------

  -- SUBUNIT --> separate (NAME) PROPER_BODY
function SUBUNIT return boolean is
begin
put(RESULT_FILE, "In subunit "); new_line(RESULT_FILE);
  if (BYPASS(TOKEN_SEPARATE)) then
    if (BYPASS(TOKEN_LEFT_PAREN)) then
      if (NAME) then
        if (BYPASS(TOKEN_RIGHT_PAREN)) then
          if (PROPER_BODY) then
            return (TRUE);
          else
            SYNTAX_ERROR("Subunit");
          end if;                 -- if proper_body statement
        else
          SYNTAX_ERROR("Subunit");
        end if;                         -- if bypass(token_right_paren)
      else
        SYNTAX_ERROR("Subunit");
      end if;                           -- if name statement
    else
      SYNTAX_ERROR("Subunit");
    end if;                             -- if bypass(token_left_paren)
  else
    return (FALSE);
  end if;                     -- if bypass(token_separate)
```

```
end SUBUNIT;

end PARSER_0;


--**********************************************************--
--
-- TITLE:        AN ADA SOFTWARE METRIC
--
-- MODULE NAME:   PACKAGE PARSER_1
-- DATE CREATED:  17 JUL 86
-- LAST MODIFIED: 30 MAY 87
--
-- AUTHORS:      LCDR JEFFREY L. NIEDER
--               LT KARL S. FAIRBANKS, JR.
--               LCDR PAUL M. HERZIG
-- DESCRIPTION:  This package contains thirty-six functions
--      that make up the top level productions for our top-down,
--      recursive descent parser. Each function is preceded
--      by the grammar productions they are implementing.
--
--**********************************************************--

with PARSER_2, PARSER_3, HENRY_GLOBAL, HENRY, BYPASS_FUNCTION,
   HALSTEAD_METRIC, GLOBAL_PARSER, GLOBAL, TEXT_IO;
use PARSER_2, PARSER_3, HENRY_GLOBAL, HENRY, BYPASS_FUNCTION,
   HALSTEAD_METRIC, GLOBAL_PARSER, GLOBAL, TEXT_IO;

package PARSER_1 is
 function GENERIC_DECLARATION return boolean;
 function GENERIC_PARAMETER_DECLARATION return boolean;
 function GENERIC_FORMAL_PART return boolean;
 function PROCEDURE_UNIT return boolean;
 function SUBPROGRAM_BODY return boolean;
 function FUNCTION_UNIT return boolean;
 function FUNCTION_UNIT_TAIL return boolean;
 function FUNCTION_BODY return boolean;
 function FUNCTION_BODY_TAIL return boolean;
 function TASK_DECLARATION return boolean;
 function TASK_BODY return boolean;
 function TASK_BODY_TAIL return boolean;
 function PACKAGE_DECLARATION return boolean;
 function PACKAGE_UNIT return boolean;
 function PACKAGE_BODY return boolean;
 function PACKAGE_BODY_TAIL return boolean;
 function PACKAGE_TAIL_END return boolean;
 function DECLARATIVE_PART return boolean;
 function BASIC_DECLARATIVE_ITEM return boolean;
 function BASIC_DECLARATION return boolean;
 function LATER_DECLARATIVE_ITEM return boolean;
 function PROPER_BODY return boolean;
```

```
 function SEQUENCE_OF_STATEMENTS return boolean;
 function STATEMENT return boolean;
 function COMPOUND_STATEMENT return boolean;
 function BLOCK_STATEMENT return boolean;
 function IF_STATEMENT return boolean;
 function CASE_STATEMENT return boolean;
 function CASE_STATEMENT_ALTERNATIVE return boolean;
 function LOOP_STATEMENT return boolean;
 function EXCEPTION_HANDLER return boolean;
 function ACCEPT_STATEMENT return boolean;
 function SELECT_STATEMENT return boolean;
 function SELECT_STATEMENT_TAIL return boolean;
 function SELECT_ALTERNATIVE return boolean;
 function SELECT_ENTRY_CALL return boolean;
end PARSER_1;

--------------------------------------------------------------------------------
--------------------------------------------------------------------------------

package body PARSER_1 is

 -- GENERIC_DECLARATION --> [GENERIC_PARAMETER_DECLARATION ?
 --                          GENERIC_FORMAL_PART
function GENERIC_DECLARATION return boolean is
begin
put(RESULT_FILE,"In generic declaration "); new_line(RESULT_FILE);
 if (GENERIC_PARAMETER_DECLARATION) then
   null;
 end if;
 if (GENERIC_FORMAL_PART) then
   return(TRUE);
 else
   return (FALSE);
 end if;
end GENERIC_DECLARATION;

--------------------------------------------------------------------------------

 -- GENERIC_PARAMETER_DECLARATION --> IDENTIFIER_LIST : [MODE ? NAME
 --                                      [:= EXPRESSION ?] ;
 --                               -->   type private [DISCRIMINANT_PART ?]
 --                                       is PRIVATE_TYPE_DECLARATION ;
 --                               -->   type private [DISCRIMINANT_PART ?]
 --                                       is GENERIC_TYPE_DEFINITION ;
 --                               --> with procedure PROCEDURE_UNIT
 --                               --> with function FUNCTION_UNIT
function GENERIC_PARAMETER_DECLARATION return boolean is
begin
put(RESULT_FILE, "In generic parameter declaration "); new_line(RESULT_FILE);
 if (IDENTIFIER_LIST) then
   if (BYPASS(TOKEN_COLON)) then
     if (MODE) then
```

```
          null;
        end if;                                   -- if mode statement
        if (NAME) then                            -- check for type_mark
          if (BYPASS(TOKEN_ASSIGNMENT)) then
            if (EXPRESSION) then
              null;
            else
              SYNTAX_ERROR("Generic parameter declaration");
            end if;                 -- if expression statement
          end if;                   -- if bypass(token_assignment)
          if (BYPASS(TOKEN_SEMICOLON)) then
            return (TRUE);
          else
            SYNTAX_ERROR("Generic parameter declaration");
          end if;                   -- if bypass(token_semicolon)
        else
          SYNTAX_ERROR("Generic parameter declaration");
        end if;                     -- if type_mark statement
      else
        SYNTAX_ERROR("Generic parameter declaration");
      end if;                       -- if bypass(token_colon)
    elsif (BYPASS(TOKEN_TYPE)) then
      if (BYPASS(TOKEN_IDENTIFIER)) then
        if (DISCRIMINANT_PART) then
          null;
        end if;                                   -- if discriminant_part
        if (BYPASS(TOKEN_IS)) then
          if (PRIVATE_TYPE_DECLARATION) then
            if (BYPASS(TOKEN_SEMICOLON)) then
              return (TRUE);
            else
              SYNTAX_ERROR("Generic parameter declaration");
            end if;                     -- if bypass(token_semicolon)
          elsif (GENERIC_TYPE_DEFINITION) then
            if (BYPASS(TOKEN_SEMICOLON)) then
              return (TRUE);
            else
              SYNTAX_ERROR("Generic parameter declaration");
            end if;                     -- if bypass(token_semicolon)
          else
            SYNTAX_ERROR("Generic parameter declaration");
          end if;                                 -- if private_type_declaration
        else
          SYNTAX_ERROR("Generic parameter declaration");
        end if;                                   -- if bypass(token_is)
      else
        SYNTAX_ERROR("Generic parameter declaration");
      end if;                                     -- if bypass(token_identifier)
    elsif (BYPASS(TOKEN_WITH)) then
      if (BYPASS(TOKEN_PROCEDURE)) then
        DECLARE_TYPE := PROCEDURE_DECLARE;
        if (PROCEDURE_UNIT) then
```

```
      return (TRUE);
    else
      SYNTAX_ERROR("Generic parameter declaration");
    end if;                              -- if procedure_unit statement
  elsif (BYPASS(TOKEN_FUNCTION)) then
    DECLARE_TYPE := FUNCTION_DECLARE;
    if (FUNCTION_UNIT) then
      return (TRUE);
    else
      SYNTAX_ERROR("Generic parameter declaration");
    end if;                              -- if function_unit statement
  else
    SYNTAX_ERROR("Generic parameter declaration");
  end if;                                -- if bypass(token_procedure)
 else
   return (FALSE);
 end if;                      -- if identifier_list
end GENERIC_PARAMETER_DECLARATION;

------------------------------------------------------------------------------

 -- GENERIC_FORMAL_PART --> procedure PROCEDURE_UNIT
 --                     --> function FUNCTION_UNIT
 --                     --> package PACKAGE_DECLARATION
function GENERIC_FORMAL_PART return boolean is
begin
put(RESULT_FILE, "In generic formal part "); new_line(RESULT_FILE);
 if (BYPASS(TOKEN_PROCEDURE)) then
   DECLARE_TYPE := PROCEDURE_DECLARE;
   if (PROCEDURE_UNIT) then
     return (TRUE);
   else
     SYNTAX_ERROR("Generic formal part");
   end if;                               -- if procedure_unit statement
 elsif (BYPASS(TOKEN_FUNCTION)) then
   DECLARE_TYPE := FUNCTION_DECLARE;
   if (FUNCTION_UNIT) then
     return (TRUE);
   else
     SYNTAX_ERROR("Generic formal part");
   end if;                               -- if function_unit statement
 elsif (BYPASS(TOKEN_PACKAGE)) then
   DECLARE_TYPE := PACKAGE_DECLARE;
   if (PACKAGE_DECLARATION) then
     return (TRUE);
   else
     SYNTAX_ERROR("Generic formal part");
   end if;                               -- if package_declaration
 else
   return (FALSE);
 end if;
end GENERIC_FORMAL_PART;
```

```
-------------------------------------------------------------------------------

 -- PROCEDURE_UNIT --> identifier [FORMAL_PART ?] is SUBPROGRAM_BODY
 --                --> identifier [FORMAL_PART ?] ;
 --                --> identifier [FORMAL_PART ?] renames NAME ;
function PROCEDURE_UNIT return boolean is
begin
put(RESULT_FILE, "In procedure unit "); new_line(RESULT_FILE);
 DECLARATION := TRUE;
   HENRY_WRITE_ENABLE := TRUE;
 if (BYPASS(TOKEN_IDENTIFIER)) then
  if PACKAGE_BODY_DECLARE then
   WRITE_HENRY_DATA(LOCAL_DECLARE, DUMMY_LEXEME,
            PROCEDURE_TYPE, NONE, LAST_RECORD);
  end if;
  SCOPE_LEVEL := SCOPE_LEVEL + 1;
  if (FORMAL_PART) then
   null;
  end if;                    -- if formal part statement
  if (BYPASS(TOKEN_IS)) then
   WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, END_PARAM_DECLARE,
            NONE, NEXT_HEN);
   CREATE_NODE(NEXT_HEN, LAST_RECORD);
   WRITE_LINE_COUNT(LAST_RECORD.NOMEN, HENRY_LINE_COUNT,
            DUMMY9s, NEXT_LINE);
   if (SUBPROGRAM_BODY) then
    WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, END_PROCEDURE_CALL,
             NONE, NEXT_HEN);
    CREATE_NODE(NEXT_HEN, LAST_RECORD);
    WRITE_LINE_COUNT(DUMMY_LEXEME, DUMMY9s, HENRY_LINE_COUNT,
             NEXT_LINE);
    CREATE_LINE_COUNT_NODE(NEXT_LINE, LAST_LINE);
    SCOPE_LEVEL := SCOPE_LEVEL - 1;
    return (TRUE);
   else
    SYNTAX_ERROR("Procedure unit");
   end if;                   -- if subprogram body statement
  elsif (BYPASS(TOKEN_SEMICOLON)) then
   SCOPE_LEVEL := SCOPE_LEVEL - 1;
   return (TRUE);
  elsif (BYPASS(TOKEN_RENAMES)) then
   if (NAME) then
    if (BYPASS(TOKEN_SEMICOLON)) then
     SCOPE_LEVEL := SCOPE_LEVEL - 1;
     return (TRUE);
    else
     SYNTAX_ERROR("Procedure unit");
    end if;                  -- if bypass(token_semicolon)
   else
    SYNTAX_ERROR("Procedure unit");
   end if;                   -- if name statement
```

```
    end if;                         -- if bypass(token_is)
  else
    return (FALSE);
  end if;                               -- if bypass(token_identifier)
end PROCEDURE_UNIT;

---------------------------------------------------------------------------

  -- SUBPROGRAM_BODY --> new NAME [GENERIC_ACTUAL_PART ?] ;
  --                 --> separate ;
  --                 --> <> ;
  --                 --> DECLARATIVE_PART ? begin SEQUENCE_OF_STATEMENTS
  --                      [exception [EXCEPTION_HANDLER]+ ?] end [DESIGNATOR ?] ;
  --                 --> NAME ;
function SUBPROGRAM_BODY return boolean is

NAME_POINTER : POINTER;

begin
put(RESULT_FILE, "In subprogram_body "); new_line(RESULT_FILE);
NAME_POINTER := NEXT_HEN;
  DECLARATION := TRUE;
  if (BYPASS(TOKEN_NEW))then
    HENRY_WRITE_ENABLE := FALSE;
    if (NAME) then
      if (GENERIC_ACTUAL_PART) then
        null;
      end if;                      -- if generic actual part
      if (BYPASS(TOKEN_SEMICOLON)) then
        return (TRUE);
      else
        SYNTAX_ERROR("Subprogram body");
      end if;                      -- if bypass(token_semicolon)
    else
      SYNTAX_ERROR("Subprogram body");
    end if;                        -- if name statement
  elsif (BYPASS(TOKEN_SEPARATE)) then
    if (BYPASS(TOKEN_SEMICOLON)) then
      return (TRUE);
    else
      SYNTAX_ERROR("Subprogram body");
    end if;                        -- if bypass(token_semicolon)
  elsif (BYPASS(TOKEN_BRACKETS)) then
    if (BYPASS(TOKEN_SEMICOLON)) then
      return (TRUE);
    else
      SYNTAX_ERROR("Subprogram body");
    end if;                        -- if bypass(token_semicolon)
  elsif (DECLARATIVE_PART) then
     WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, END_DECLARATIONS,
               NONE, NEXT_HEN);
     CREATE_NODE(NEXT_HEN, LAST_RECORD);
```

```
if (BYPASS(TOKEN_BEGIN)) then
  DECLARATION := FALSE;
  if (SEQUENCE_OF_STATEMENTS) then
    if (BYPASS(TOKEN_EXCEPTION)) then
      if (EXCEPTION_HANDLER) then
        while (EXCEPTION_HANDLER) loop
          null;
        end loop;
      else
        SYNTAX_ERROR("Subprogram body");
      end if;                       -- if exception_handler statement
    end if;                               -- if bypass(token_exception)
    if (BYPASS(TOKEN_END)) then
        HENRY_WRITE_ENABLE := FALSE;
      if (DESIGNATOR) then
        null;
      end if;                     -- if designator statement
      if (BYPASS(TOKEN_SEMICOLON)) then
        DECLARATION := TRUE;
        return (TRUE);
      else
        SYNTAX_ERROR("Subprogram body");
      end if;                     -- if bypass(token_semicolon)
    else
      SYNTAX_ERROR("Subprogram body");
    end if;                       -- if bypass(token_end)
  else
    SYNTAX_ERROR("Subprogram body");
  end if;                         -- if sequence of statements
else
  SYNTAX_ERROR("Subprogram body");
end if;                           -- if bypass(token_begin)
elsif (BYPASS(TOKEN_BEGIN)) then
  DECLARATION := FALSE;
  WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, END_DECLARATIONS,
          NONE, NEXT_HEN);
  CREATE_NODE(NEXT_HEN, LAST_RECORD);
  if (SEQUENCE_OF_STATEMENTS) then
    if (BYPASS(TOKEN_EXCEPTION)) then
      if (EXCEPTION_HANDLER) then
        while (EXCEPTION_HANDLER) loop
          null;
        end loop;
      else
        SYNTAX_ERROR("Subprogram body");
      end if;                               -- if exception_handler statement
    end if;                                 -- if bypass(token_exception)
    if (BYPASS(TOKEN_END)) then
      HENRY_WRITE_ENABLE := FALSE;
      if (DESIGNATOR) then
        null;
      end if;                     -- if designator statement
```

```
        if (BYPASS(TOKEN_SEMICOLON)) then
          DECLARATION := TRUE;
          return (TRUE);
        else
          SYNTAX_ERROR("Subprogram body");
        end if;                   -- if bypass(token_semicolon)
      else
        SYNTAX_ERROR("Subprogram body");
      end if;                     -- if bypass(token_end)
    else
      SYNTAX_ERROR("Subprogram body");
    end if;                       -- if sequence of statements
  elsif (NAME) then
    if (BYPASS(TOKEN_SEMICOLON)) then
      return (TRUE);
    else
      SYNTAX_ERROR("Subprogram body");
    end if;                       -- if bypass(token_semicolon)
  else
    return (FALSE);
  end if;                         -- if bypass(token_new)
end SUBPROGRAM_BODY;

--------------------------------------------------------------------------------

  -- FUNCTION_UNIT --> DESIGNATOR FUNCTION_UNIT_TAIL
function FUNCTION_UNIT return boolean is
begin
put(RESULT_FILE, "In function unit "); new_line(RESULT_FILE);
 DECLARATION := TRUE;
 HENRY_WRITE_ENABLE := TRUE;
 if (DESIGNATOR) then
   if PACKAGE_BODY_DECLARE then
      WRITE_HENRY_DATA(LOCAL_DECLARE, DUMMY_LEXEME, FUNCTION_TYPE,
                NONE, LAST_RECORD);
      WRITE_LINE_COUNT(LAST_RECORD.NOMEN, HENRY_LINE_COUNT,
                DUMMY9s, NEXT_LINE);
   end if;
   SCOPE_LEVEL := SCOPE_LEVEL + 1;
   if (FUNCTION_UNIT_TAIL) then

     SCOPE_LEVEL := SCOPE_LEVEL - 1;
     return (TRUE);
   else
     SYNTAX_ERROR("Function unit");
   end if;
 else
   return (FALSE);
 end if;
end FUNCTION_UNIT;

--------------------------------------------------------------------------------
```

```
 -- FUNCTION_UNIT_TAIL --> is new NAME [GENERIC_ACTUAL_PART ?] ;
 --                    --> [FORMAL_PART ?] return NAME FUNCTION_BODY
function FUNCTION_UNIT_TAIL return boolean is
begin
put(RESULT_FILE, "In function unit tail "); new_line(RESULT_FILE);
 if (BYPASS(TOKEN_IS)) then
    FUNCTION_PARAM_DECLARE := TRUE;
   if (BYPASS(TOKEN_NEW)) then
    if (NAME) then
     if (GENERIC_ACTUAL_PART) then
       null;
     end if;              -- if generic actual part
     if (BYPASS(TOKEN_SEMICOLON)) then
       return (TRUE);
     else
       SYNTAX_ERROR("Function unit tail");
     end if;              -- if bypass(token_semicolon)
    else
     SYNTAX_ERROR("Function unit tail");
    end if;               -- if name statement
   else
    SYNTAX_ERROR("Function unit tail");
   end if;                -- if bypass(token_new)
 elsif (FORMAL_PART) then
   FUNCTION_PARAM_DECLARE := FALSE;
  if (BYPASS(TOKEN_RETURN)) then
   if (NAME) then                    -- check for type_mark
    if (FUNCTION_BODY) then
      return (TRUE);
    else
      SYNTAX_ERROR("Function unit tail");
    end if;              -- if function body statement
   else
    SYNTAX_ERROR("Function unit tail");
   end if;               -- if type mark statement
  else
   SYNTAX_ERROR("Function unit tail");
  end if;                -- if bypass(token_return)
 elsif (BYPASS(TOKEN_RETURN)) then
  if (NAME) then                     -- check for type_mark
   if (FUNCTION_BODY) then
     return (TRUE);
   else
     SYNTAX_ERROR("Function unit tail");
   end if;               -- if function body statement
  else
   SYNTAX_ERROR("Function unit tail");
  end if;                -- if type mark statement
 else
  return (FALSE);
 end if;                 -- if bypass(token_is)
end FUNCTION_UNIT_TAIL;
```

```
-------------------------------------------------------------------------

 -- FUNCTION_BODY --> is [FUNCTION_BODY_TAIL ?
 --                --> ;
function FUNCTION_BODY return boolean is
begin
put(RESULT FILE, "In function body "); new_line(RESULT FILE);
 if (BYPASS(TOKEN_IS)) then
   WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, END_PARAM_DECLARE, NONE, NEXT_H
   CREATE_NODE(NEXT_HEN, LAST_RECORD);
   if (FUNCTION_BODY_TAIL) then
     WRITE_LINE_COUNT(DUMMY_LEXEME, DUMMY9s, HENRY_LINE_COUNT,
            NEXT_LINE);
     CREATE_LINE_COUNT_NODE(NEXT_LINE, LAST_LINE);
     WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, END_FUNCTION_TYPE,
            NONE, NEXT_HEN);
     CREATE_NODE(NEXT_HEN, LAST_RECORD);
   end if;
   return (TRUE);
 elsif (BYPASS(TOKEN_SEMICOLON)) then
   return (TRUE);
 else
   return (FALSE);
 end if;
end FUNCTION_BODY;

-------------------------------------------------------------------------

 -- FUNCTION_BODY_TAIL --> separate ;
 --                     --> <> ;
 --                     --> SUBPROGRAM_BODY
 --                     --> NAME ;
function FUNCTION_BODY_TAIL return boolean is
begin
put(RESULT_FILE, "In function body tail "); new_line(RESULT_FILE);
 if (BYPASS(TOKEN_SEPARATE)) then
   if (BYPASS(TOKEN_SEMICOLON)) then
     return (TRUE);
   else
     SYNTAX_ERROR("Function body tail");
   end if;                    -- if bypass(token_semicolon)
 elsif (BYPASS(TOKEN_BRACKETS)) then
   if (BYPASS(TOKEN_SEMICOLON)) then
     return (TRUE);
   else
     SYNTAX_ERROR("Function body tail");
   end if;                    -- if bypass(token_semicolon)
 elsif (SUBPROGRAM_BODY) then
   return (TRUE);
 elsif (NAME) then
   if (BYPASS(TOKEN_SEMICOLON)) then
     return (TRUE);
```

```
  else
   SYNTAX_ERROR("Function body tail");
  end if;                    -- if bypass(token_semicolon)
 else
  return (FALSE);
 end if;                     -- if bypass(token_separate)
end FUNCTION_BODY_TAIL;

------------------------------------------------------------------------------

 -- TASK_DECLARATION --> body TASK_BODY ;
 --                  --> [type ?] identifier [is [ENTRY_DECLARATION *
 --                          REPRESENTATION_CLAUSE]* end [identifier ?] ?] ;
function TASK_DECLARATION return boolean is
begin
put(RESULT_FILE, "In task_declaration "); new_line(RESULT_FILE);
 DECLARATION := TRUE;
 if (BYPASS(TOKEN_TYPE)) then
  null;
 end if;                          -- if bypass(token_type)
 if (BYPASS(TOKEN_BODY)) then
  if (TASK_BODY) then
   if (BYPASS(TOKEN_SEMICOLON)) then
    return (TRUE);
   else
    SYNTAX_ERROR("Task declaration");
   end if;
  else
   SYNTAX_ERROR("Task declaration");
  end if;                            -- if task_body statement
 elsif (BYPASS(TOKEN_IDENTIFIER)) then
  SCOPE_LEVEL := SCOPE_LEVEL + 1;
  if (BYPASS(TOKEN_IS)) then
   while (ENTRY_DECLARATION) loop
    null;
   end loop;
   while (REPRESENTATION_CLAUSE) loop
    null;
   end loop;
   if (BYPASS(TOKEN_END)) then
    if (BYPASS(TOKEN_IDENTIFIER)) then
     null;
    end if;                        -- if bypass(token_identifier)
    if (BYPASS(TOKEN_SEMICOLON)) then
     SCOPE_LEVEL := SCOPE_LEVEL - 1;
     return (TRUE);
    else
     SYNTAX_ERROR("Task declaration");
    end if;                        -- if bypass(token_semicolon)
   else
    SYNTAX_ERROR("Task declaration");
   end if;                         -- if bypass(token_end)
```

```
    elsif (BYPASS(TOKEN_SEMICOLON)) then
      SCOPE_LEVEL := SCOPE_LEVEL - 1;
      return (TRUE);
    else
      SYNTAX_ERROR("Task declaration");
    end if;                             -- if bypass(token_is)
  else
    return (FALSE);
  end if;                       -- if bypass(token_body)
end TASK_DECLARATION;

----------------------------------------------------------------------------

  -- TASK_BODY --> identifier is TASK_BODY_TAIL
function TASK_BODY return boolean is
begin
put(RESULT_FILE, "In task_body "); new_line(RESULT_FILE);
  if (BYPASS(TOKEN_IDENTIFIER)) then
    SCOPE_LEVEL := SCOPE_LEVEL + 1;
    if (BYPASS(TOKEN_IS)) then
      if (TASK_BODY_TAIL) then
        SCOPE_LEVEL := SCOPE_LEVEL - 1;
        return (TRUE);
      else
        SYNTAX_ERROR("Task body");
      end if;                           -- if task_body_tail statement
    else
      SYNTAX_ERROR("Task body");
    end if;                             -- if bypass(token_is)
  else
    return (FALSE);
  end if;                       -- if bypass(token_identifier)
end TASK_BODY;

----------------------------------------------------------------------------

  -- TASK_BODY_TAIL --> separate
  --                --> [DECLARATIVE_PART ?] begin SEQUENCE_OF_STATEMENTS
  --                    [exception EXCEPTION_HANDLER + ?] end [identifier ?]
function TASK_BODY_TAIL return boolean is
begin
put(RESULT_FILE, "In task_body_tail "); new_line(RESULT_FILE);
  DECLARATION := TRUE;
  if (BYPASS(TOKEN_SEPARATE)) then
    return (TRUE);
  elsif (DECLARATIVE_PART) then
    if (BYPASS(TOKEN_BEGIN)) then
      DECLARATION := FALSE;
      if (SEQUENCE_OF_STATEMENTS) then
        if (BYPASS(TOKEN_EXCEPTION)) then
          if (EXCEPTION_HANDLER) then
            while (EXCEPTION_HANDLER) loop
```

```
          null;
        end loop;
      else
        SYNTAX_ERROR("Task body tail");
      end if;                    -- if exception_handler statement
    end if;                          -- if bypass(token_exception)
    if (BYPASS(TOKEN_END)) then
      if (BYPASS(TOKEN_IDENTIFIER)) then
        null;
      end if;                    -- if bypass(token_identifier)
      DECLARATION := TRUE;
      return (TRUE);
    else
      SYNTAX_ERROR("Task body tail");
    end if;                      -- if bypass(token_end)
  else
    SYNTAX_ERROR("Task body tail");
  end if;                        -- if sequence_of_statements
else
  SYNTAX_ERROR("Task body tail");
end if;                          -- if bypass(token_begin)
elsif (BYPASS(TOKEN_BEGIN)) then
  DECLARATION := FALSE;
  if (SEQUENCE_OF_STATEMENTS) then
    if (BYPASS(TOKEN_EXCEPTION)) then
      if (EXCEPTION_HANDLER) then
        while (EXCEPTION_HANDLER) loop
          null;
        end loop;
      else
        SYNTAX_ERROR("Task body tail");
      end if;                        -- if exception_handler statement
    end if;                          -- if bypass(token_exception)
    if (BYPASS(TOKEN_END)) then
      if (BYPASS(TOKEN_IDENTIFIER)) then
        null;
      end if;                    -- if bypass(token_identifier)
      DECLARATION := TRUE;
      return (TRUE);
    else
      SYNTAX_ERROR("Task body tail");
    end if;                      -- if bypass(token_end)
  else
    SYNTAX_ERROR("Task body tail");
  end if;                        -- if sequence_of_statements
else
  return (FALSE);
end if;                          -- if bypass(token_separate)
end TASK_BODY_TAIL;

----------------------------------------------------------------------------
```

```
-- PACKAGE_DECLARATION --> body PACKAGE_BODY
--                      --> identifier PACKAGE_UNIT
function PACKAGE_DECLARATION return boolean is
begin
put(RESULT_FILE, "In package_declaration "); new_line(RESULT_FILE);
 DECLARATION := TRUE;
 HENRY_WRITE_ENABLE := TRUE;
 if (BYPASS(TOKEN_BODY)) then
  PACKAGE_BODY_DECLARE := TRUE;
  HENRY_WRITE_ENABLE  := FALSE;
  if (PACKAGE_BODY) then
   return (TRUE);
  else
   SYNTAX_ERROR("Package declaration");
  end if;                -- if package unit statement
 elsif (BYPASS(TOKEN_IDENTIFIER)) then
  WRITE_HENRY_DATA(LOCAL_DECLARE, DUMMY_LEXEME, PACKAGE_TYPE,
          NONE, LAST_RECORD);
  SCOPE_LEVEL := SCOPE_LEVEL + 1;
  if (PACKAGE_UNIT) then
   SCOPE_LEVEL := SCOPE_LEVEL - 1;
   return (TRUE);
  else
   SYNTAX_ERROR("Package declaration");
  end if;                          -- if package_unit_tail statement
 else
  return (FALSE);
 end if;                 -- if bypass(token_package)
end PACKAGE_DECLARATION;

-------------------------------------------------------------------------------

 -- PACKAGE_BODY --> identifier is PACKAGE_BODY_TAIL
function PACKAGE_BODY return boolean is
begin
put(RESULT_FILE, "In package_body "); new_line(RESULT_FILE);

 if (BYPASS(TOKEN_IDENTIFIER)) then
  SCOPE_LEVEL := SCOPE_LEVEL + 1;
  if (BYPASS(TOKEN_IS)) then
   if (PACKAGE_BODY_TAIL) then
    WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, END_PACKAGE_TYPE,
           NONE, NEXT_HEN);
    SCOPE_LEVEL := SCOPE_LEVEL - 1;
    return (TRUE);
   else
    SYNTAX_ERROR("Package body");
   end if;            -- if package_body_tail statement
  else
   SYNTAX_ERROR("Package body");
  end if;             -- if bypass(token_is)
 else
```

```
   return (FALSE);
  end if;                    -- if bypass(token_identifier)
end PACKAGE_BODY;

-----------------------------------------------------------------------------

  -- PACKAGE_BODY_TAIL --> separate ;
  --                   --> [DECLARATIVE_PART ?] [begin SEQUENCE_OF_STATEMENTS
  --                       [exception [EXCEPTION_HANDLER ~ ?] ?]
  --                         end [identifier ?] ;
function PACKAGE_BODY_TAIL return boolean is
begin
put(RESULT_FILE, "In package_body_tail "); new_line(RESULT_FILE);
 DECLARATION := TRUE;
 if (BYPASS(TOKEN_SEPARATE)) then
  if (BYPASS(TOKEN_SEMICOLON)) then
   return (TRUE);
  else
   SYNTAX_ERROR("Package body tail");
  end if;                    -- if bypass(token_semicolon)
 elsif (DECLARATIVE_PART) then
  DECLARATION := FALSE;
  if (BYPASS(TOKEN_BEGIN)) then
   if (SEQUENCE_OF_STATEMENTS) then
    if (BYPASS(TOKEN_EXCEPTION)) then
     if (EXCEPTION_HANDLER) then
      while (EXCEPTION_HANDLER) loop
       null;
      end loop;
     else
      SYNTAX_ERROR("Package body tail");
     end if;                 -- if exception_handler statement
    end if;                  -- if bypass(token_exception)
    if (BYPASS(TOKEN_END)) then
      HENRY_WRITE_ENABLE := FALSE;
      if (BYPASS(TOKEN_IDENTIFIER)) then
        null;
     end if;                 -- if bypass(token_identifier)
     if (BYPASS(TOKEN_SEMICOLON)) then
      DECLARATION := TRUE;
      return (TRUE);
     else
      SYNTAX_ERROR("Package body tail");
     end if;                 -- if bypass(token_semicolon)
    else
     SYNTAX_ERROR("Package body tail");
    end if;                  -- if bypass(token_end)
   else
    SYNTAX_ERROR("Package body tail");
   end if;                   -- if sequence_of_statements
  elsif (BYPASS(TOKEN_END)) then
    HENRY_WRITE_ENABLE := FALSE;
```

```
    if (BYPASS(TOKEN_IDENTIFIER)) then
      null;
    end if;                       -- if bypass(token_identifier)
    if (BYPASS(TOKEN_SEMICOLON)) then
      DECLARATION := TRUE;
      return (TRUE);
    else
      SYNTAX_ERROR("Package body tail");
    end if;                       -- if bypass(token_semicolon)
  else
    SYNTAX_ERROR("Package body tail");
  end if;                         -- if bypass(token_begin)
elsif (BYPASS(TOKEN_BEGIN)) then
  DECLARATION := FALSE;
  if (SEQUENCE_OF_STATEMENTS) then
    if (BYPASS(TOKEN_EXCEPTION)) then
      if (EXCEPTION_HANDLER) then
        while (EXCEPTION_HANDLER) loop
          null;
        end loop;
      else
        SYNTAX_ERROR("Package body tail");
      end if;                           -- if exception_handler statement
    end if;                             -- if bypass(token_exception)
    if (BYPASS(TOKEN_END)) then
      HENRY_WRITE_ENABLE := FALSE;
      if (BYPASS(TOKEN_IDENTIFIER)) then
        null;
      end if;                     -- if bypass(token_identifier)
      if (BYPASS(TOKEN_SEMICOLON)) then
        DECLARATION := TRUE;
        return (TRUE);
      else
        SYNTAX_ERROR("Package body tail");
      end if;                     -- if bypass(token_semicolon)
    else
      SYNTAX_ERROR("Package body tail");
    end if;                       -- if bypass(token_end)
  else
    SYNTAX_ERROR("Package body tail");
  end if;                         -- if sequence_of_statements
elsif (BYPASS(TOKEN_END)) then
  HENRY_WRITE_ENABLE := FALSE;
  if (BYPASS(TOKEN_IDENTIFIER)) then
    null;
  end if;                         -- if bypass(token_identifier)
  if (BYPASS(TOKEN_SEMICOLON)) then
    return (TRUE);
  else
    SYNTAX_ERROR("Package body tail");
  end if;                         -- if bypass(token_semicolon)
else
```

```
  return (FALSE);
 end if;                        -- if bypass(token_separate)
end PACKAGE_BODY_TAIL;

------------------------------------------------------------------------

 -- PACKAGE_UNIT --> is PACKAGE_TAIL_END
 --              --> renames NAME ;
function PACKAGE_UNIT return boolean is
begin
put(RESULT_FILE, "In package_unit "); new_line(RESULT_FILE);
 if (BYPASS(TOKEN_IS)) then
  if (PACKAGE_TAIL_END) then
   return (TRUE);
  else
   SYNTAX_ERROR("Package unit");
  end if;
 elsif (BYPASS(TOKEN_RENAMES)) then
  if (NAME) then
   if (BYPASS(TOKEN_SEMICOLON)) then
    return (TRUE);
   else
    SYNTAX_ERROR("Package unit");
   end if;                       -- if bypass(token_semicolon)
  else
   SYNTAX_ERROR("Package unit");
  end if;                       -- if name statement
 else
  return (FALSE);
 end if;                        -- if bypass(token_is)
end PACKAGE_UNIT;

------------------------------------------------------------------------

 -- PACKAGE_TAIL_END --> new NAME [GENERIC_ACTUAL_PART ?] ;
 --                  --> [BASIC_DECLARATIVE_ITEM]* [private
 --                        [BASIC_DECLARATIVE_ITEM]* ?] end [identifier ?] ;
function PACKAGE_TAIL_END return boolean is
begin
put(RESULT_FILE,"In package_tail_end "); new_line(RESULT_FILE);
 if (BYPASS(TOKEN_NEW)) then
  if (NAME) then
   if (GENERIC_ACTUAL_PART) then
    null;
   end if;                      -- if generic_actual_part statement
   if (BYPASS(TOKEN_SEMICOLON)) then
    return (TRUE);
   else
    SYNTAX_ERROR("Package tail end");
   end if;                      -- if bypass(token_semicolon)
  else
   SYNTAX_ERROR("Package tail end");
```

```
  end if;                         -- if name statement
elsif (BASIC_DECLARATIVE_ITEM) then
  while (BASIC_DECLARATIVE_ITEM) loop
    null;
  end loop;
  if (BYPASS(TOKEN_PRIVATE)) then
    while (BASIC_DECLARATIVE_ITEM) loop
      null;
    end loop;
  end if;                               -- if bypass(token_private)
  if (BYPASS(TOKEN_END)) then
    HENRY_WRITE_ENABLE := FALSE;
    if (BYPASS(TOKEN_IDENTIFIER)) then
      null;
    end if;
    if (BYPASS(TOKEN_SEMICOLON)) then
      WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, END_PACKAGE_DECLARE,
                NONE, NEXT_HEN);
      CREATE_NODE(NEXT_HEN, LAST_RECORD);
      return (TRUE);
    else
      SYNTAX_ERROR("Package tail end");
    end if;                       -- if bypass(token_semicolon)
  else
    SYNTAX_ERROR("Package tail end");
  end if;                         -- if bypass(token_end)
elsif (BYPASS(TOKEN_PRIVATE)) then
  while (BASIC_DECLARATIVE_ITEM) loop
    null;
  end loop;
  if (BYPASS(TOKEN_END)) then
    HENRY_WRITE_ENABLE := FALSE;
    if (BYPASS(TOKEN_IDENTIFIER)) then
      null;
    end if;
    if (BYPASS(TOKEN_SEMICOLON)) then
      WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, END_PACKAGE_DECLARE,
                NONE, NEXT_HEN);
      CREATE_NODE(NEXT_HEN, LAST_RECORD);
      return (TRUE);
    else
      SYNTAX_ERROR("Package tail end");
    end if;                       -- if bypass(token_semicolon)
  else
    SYNTAX_ERROR("Package tail end");
  end if;                         -- if bypass(token_end)
elsif (BYPASS(TOKEN_END)) then
  HENRY_WRITE_ENABLE := FALSE;
  if (BYPASS(TOKEN_IDENTIFIER)) then
    null;
  end if;
  if (BYPASS(TOKEN_SEMICOLON)) then
```

```
    WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, END_PACKAGE_DECLARE,
                  NONE, NEXT_HEN);
    CREATE_NODE(NEXT_HEN, LAST_RECORD);
   return (TRUE);
  else
   SYNTAX_ERROR("Package tail end");
  end if;                           -- if bypass(token_semicolon)
 else
  return (FALSE);
 end if;                            -- if bypass(token_new)
end PACKAGE_TAIL_END;

-----------------------------------------------------------------------------

 -- BASIC_DECLARATIVE_ITEM --> BASIC_DECLARATIVE
 --                        --> REPRESENTATION_CLAUSE
 --                        --> use WITH_OR_USE_CLAUSE
function BASIC_DECLARATIVE_ITEM return boolean is
begin
put(RESULT_FILE, "In basic_declarative_item "); new_line(RESULT_FILE);
HENRY_WRITE_ENABLE := TRUE;
 if (BASIC_DECLARATION) then
  HENRY_WRITE_ENABLE := FALSE;
  return (TRUE);
 elsif (REPRESENTATION_CLAUSE) then
  return (TRUE);
 elsif (BYPASS(TOKEN_USE)) then
  if (WITH_OR_USE_CLAUSE) then
   return (TRUE);
  else
   SYNTAX_ERROR("Basic declarative item");
  end if;
 else
  return (FALSE);
 end if;
end BASIC_DECLARATIVE_ITEM;

-----------------------------------------------------------------------------

 -- DECLARATIVE_PART--> {BASIC_DECLARATIVE_ITEM}* {LATER_DECLARATIVE_ITEM}*
function DECLARATIVE_PART return boolean is
begin
put(RESULT_FILE, "In declarative_part "); new_line(RESULT_FILE);
 while (BASIC_DECLARATIVE_ITEM) loop
  null;
 end loop;
 while (LATER_DECLARATIVE_ITEM) loop
  null;
 end loop;
 return (TRUE);
end DECLARATIVE_PART;
```

```
------------------------------------------------------------------------

-- BASIC_DECLARATION --> type TYPE_DECLARATION
--                   --> subtype SUBTYPE_DECLARATION
--                   --> procedure PROCEDURE_UNIT
--                   --> function FUNCTION_UNIT
--                   --> package PACKAGE_DECLARATION
--                   --> generic GENERIC_DECLARATION
--                   --> IDENTIFIER_DECLARATION
--                   --> task TASK_DECLARATION
function BASIC_DECLARATION return boolean is
begin
put(RESULT_FILE, "In basic_declaration "); new_line(RESULT_FILE);
 if (BYPASS(TOKEN_TYPE)) then
  if (TYPE_DECLARATION) then
   return (TRUE);
  else
   SYNTAX_ERROR("Basic declaration");
  end if;
 elsif (BYPASS(TOKEN_SUBTYPE)) then
  if (SUBTYPE_DECLARATION) then
   return (TRUE);
  else
   SYNTAX_ERROR("Basic declaration");
  end if;
 elsif (BYPASS(TOKEN_PROCEDURE)) then
  DECLARE_TYPE := PROCEDURE_DECLARE;
  if (PROCEDURE_UNIT) then
  HENRY_WRITE_ENABLE := FALSE;
   return (TRUE);
  else
   SYNTAX_ERROR("Basic declaration");
  end if;                        -- if procedure_unit statement
 elsif (BYPASS(TOKEN_FUNCTION)) then
  DECLARE_TYPE := FUNCTION_DECLARE;
  if (FUNCTION_UNIT) then
  HENRY_WRITE_ENABLE := FALSE;
   return (TRUE);
  else
   SYNTAX_ERROR("Basic declaration");
  end if;                        -- if function_unit statement
 elsif (BYPASS(TOKEN_PACKAGE)) then
  DECLARE_TYPE := PACKAGE_DECLARE;
  if (PACKAGE_DECLARATION) then
   return (TRUE);
  else
   SYNTAX_ERROR("Basic declaration");
  end if;                        -- if package_declaration
 elsif (BYPASS(TOKEN_GENERIC)) then
  if (GENERIC_DECLARATION) then
   return (TRUE);
  else
```

```
    SYNTAX_ERROR("Basic declaration");
   end if;                              -- if generic declaration
  elsif (IDENTIFIER_DECLARATION) then
   HENRY_WRITE_ENABLE := FALSE;
   return (TRUE);
  elsif (BYPASS(TOKEN_TASK)) then
   DECLARE_TYPE := TASK_DECLARE;
   if (TASK_DECLARATION) then
    return (TRUE);
   else
    SYNTAX_ERROR("Basic declaration");
   end if;
  else
   return (FALSE);
  end if;
end BASIC_DECLARATION;

-----------------------------------------------------------------------

  -- LATER_DECLARATIVE_ITEM --> PROPER_BODY
  --                          --> generic GENERIC_DECLARATION
  --                          --> use WITH_OR_USE_CLAUSE
function LATER_DECLARATIVE_ITEM return boolean is
begin
put(RESULT_FILE, "In later_declarative_item "); new_line(RESULT_FILE);
  if (PROPER_BODY) then                      -- check for body_declaration
   return (TRUE);
  elsif (BYPASS(TOKEN_GENERIC)) then
   if (GENERIC_DECLARATION) then
    return (TRUE);
   else
    SYNTAX_ERROR("Later declarative item");
   end if;                              -- if generic_declaration
  elsif (BYPASS(TOKEN_USE)) then
   if (WITH_OR_USE_CLAUSE) then
    return (TRUE);
   else
    SYNTAX_ERROR("Later declarative item");
   end if;                              -- if with or use clause
  else
   return (FALSE);
  end if;
end LATER_DECLARATIVE_ITEM;

-----------------------------------------------------------------------

  -- PROPER_BODY --> procedure PROCEDURE_UNIT
  --              --> function FUNCTION_UNIT
  --              --> package PACKAGE_DECLARATION
  --              --> task TASK_DECLARATION
function PROPER_BODY return boolean is
begin
```

```
put(RESULT_FILE, "In proper_body "); new_line(RESULT_FILE);
 if (BYPASS(TOKEN_PROCEDURE)) then
   DECLARE_TYPE := PROCEDURE_DECLARE;
   if (PROCEDURE_UNIT) then
    return (TRUE);
   else
    SYNTAX_ERROR("Proper body");
   end if;                          -- if procedure_unit statement
 elsif (BYPASS(TOKEN_FUNCTION)) then
   DECLARE_TYPE := FUNCTION_DECLARE;
   if (FUNCTION_UNIT) then
    return (TRUE);
   else
    SYNTAX_ERROR("Proper body");
   end if;                          -- if function_unit statement
 elsif (BYPASS(TOKEN_PACKAGE)) then
   DECLARE_TYPE := PACKAGE_DECLARE;
   if (PACKAGE_DECLARATION) then
    return (TRUE);
   else
    SYNTAX_ERROR("Proper body");
   end if;                          -- if package_declaration
 elsif (BYPASS(TOKEN_TASK)) then
   DECLARE_TYPE := TASK_DECLARE;
   if (TASK_DECLARATION) then
    return (TRUE);
   else
    SYNTAX_ERROR("Proper body");
   end if;
 else
   return (FALSE);
 end if;                    -- if bypass(token_procedure)
end PROPER_BODY;

-------------------------------------------------------------------------------

 -- SEQUENCE_OF_STATEMENTS --> STATEMENT -
function SEQUENCE_OF_STATEMENTS return boolean is
begin
put(RESULT_FILE, "In sequence_of_statements "); new_line(RESULT_FILE);
 if (STATEMENT) then
   while (STATEMENT) loop
    null;
   end loop;
   return (TRUE);
 else
   return (FALSE);
 end if;
end SEQUENCE_OF_STATEMENTS;

-------------------------------------------------------------------------------
```

```
-- STATEMENT --> LABEL ? SIMPLE_STATEMENT
--            --> LABEL ? COMPOUND_STATEMENT
function STATEMENT return boolean is
begin
put(RESULT_FILE, "In statement "); new_line(RESULT_FILE);
 if (LABEL) then
   null;
 end if;
 if (SIMPLE_STATEMENT) then
   return (TRUE);
 elsif (COMPOUND_STATEMENT) then
   return (TRUE);
 else
   return (FALSE);
 end if;
end STATEMENT;

---------------------------------------------------------------------------

-- COMPOUND_STATEMENT --> if IF_STATEMENT
--                    --> case CASE_STATEMENT
--                    --> LOOP_STATEMENT
--                    --> BLOCK_STATEMENT
--                    --> accept ACCEPT_STATEMENT
--                    --> select SELECT_STATEMENT
function COMPOUND_STATEMENT return boolean is
begin
put(RESULT_FILE, "In compound statement "); new_line(RESULT_FILE);
 if (BYPASS(TOKEN_IF)) then
   NESTING_METRIC(IF_CONSTRUCT);
   if (IF_STATEMENT) then
     return (TRUE);
   else
     SYNTAX_ERROR("Compound statement");
   end if;                           -- if if_statement
 elsif (BYPASS(TOKEN_CASE)) then
   NESTING_METRIC(CASE_CONSTRUCT);
   if (CASE_STATEMENT) then
     return (TRUE);
   else
     SYNTAX_ERROR("Compound statement");
   end if;                           -- if case_statement
 elsif (LOOP_STATEMENT) then
   return (TRUE);
 elsif (BLOCK_STATEMENT) then
   return (TRUE);
 elsif (BYPASS(TOKEN_ACCEPT)) then
   if (ACCEPT_STATEMENT) then
     return (TRUE);
   else
     SYNTAX_ERROR("Compound statement");
   end if;
```

```
 elsif (BYPASS(TOKEN SELECT)) then
  if (SELECT STATEMENT) then
   return (TRUE);
  else
   SYNTAX ERROR("Compound statement");
  end if;
 else
  return (FALSE);
 end if;
end COMPOUND STATEMENT;

--------------------------------------------------------------------------

 -- BLOCK STATEMENT --> [identifier : ?  declare DECLARATIVE_PART ?]
 --                     begin SEQUENCE_OF_STATEMENTS [exception
 --                     EXCEPTION HANDLER - ?] ?] end [identifier ?] ;
function BLOCK STATEMENT return boolean is
 DECLARE STATUS : boolean;
begin
put(RESULT FILE, "In block_statement "); new_line(RESULT_FILE);
 if (DECLARATION) then
  DECLARE STATUS := TRUE;
 else
  DECLARATION := TRUE;
  DECLARE STATUS : FALSE;
 end if;
 DECLARE TYPE := BLOCK_DECLARE;
 if (BYPASS(TOKEN IDENTIFIER)) then
  SCOPE LEVEL := SCOPE LEVEL + 1;
  if (BYPASS(TOKEN COLON)) then
   SCOPE LEVEL := SCOPE_LEVEL - 1;
  else
   SYNTAX_ERROR("Block statement");
  end if;                         -- if bypass(token_colon)
 else
  DECLARE_TYPE := VARIABLE_DECLARE;
 end if;                  -- if bypass(token_identifier)
 if (BYPASS(TOKEN DECLARE)) then
  SCOPE LEVEL := SCOPE LEVEL + 1;
  if (DECLARATIVE PART) then
   null;
  else
   SYNTAX_ERROR("Block statement");
  end if;                         -- if declarative part statement
 end if;                  -- if bypass(token declare)
 if (BYPASS(TOKEN BEGIN)) then
  DECLARATION := FALSE;
  if (SEQUENCE_OF STATEMENTS) then
   if (BYPASS(TOKEN EXCEPTION)) then
    if (EXCEPTION HANDLER) then
     while (EXCEPTION HANDLER) loop
      null;
```

```
        end loop;
      else
        SYNTAX ERROR("Block statement");
      end if;                            -- if exception handler statement
    end if                               -- if bypass(token exception)
    if (BYPASS(TOKEN END)) then
      if (BYPASS(TOKEN IDENTIFIER)) then
        null;
      end if;                            -- if bypass(token identifier)
      if (BYPASS(TOKEN SEMICOLON)) then
        SCOPE LEVEL := SCOPE LEVEL - 1;
        DECLARATION := TRUE;
        return (TRUE);
      else
        SYNTAX ERROR("Block statement");
      end if;                            -- if bypass(token semicolon)
    else
      SYNTAX ERROR("Block statement");
    end if                               -- if bypass(token end)
  else
    SYNTAX ERROR("Block statement");
  end if;                                -- if sequence of statements
else
  if not (DECLARE STATUS) then
    DECLARATION := FALSE;
  end if;
  return (FALSE);
end if;                          -- if bypass(token begin)
end BLOCK STATEMENT;

------------------------------------------------------------------------

 -- IF STATEMENT -- ::= EXPRESSION then SEQUENCE OF STATEMENTS
 --                    elsif EXPRESSION then SEQUENCE OF STATEMENTS }
 --                    else SEQUENCE OF STATEMENTS ] end if ;
function IF STATEMENT return boolean is
begin
put(RESULT FILE, "In if statement "); new line(RESULT FILE);
 if (EXPRESSION) then
  if (BYPASS(TOKEN THEN)) then
   if (SEQUENCE OF STATEMENTS) then
    while (BYPASS(TOKEN ELSIF)) loop
     if (EXPRESSION) then
      if (BYPASS(TOKEN THEN)) then
       if not (SEQUENCE OF STATEMENTS) then
        SYNTAX ERROR("If statement");
       end if;                -- if not sequence of statements
      else
       SYNTAX ERROR("If statement");
      end if;                 -- if bypass(token then)
     else
      SYNTAX ERROR("If statement");
```

```
        end if;                      -- if expression statement
      end loop;
      if (BYPASS(TOKEN_ELSE)) then
        if (SEQUENCE_OF_STATEMENTS) then
          null;
        else
          SYNTAX_ERROR("If statement");
        end if;                      -- if sequence_of_statements
      end if;                        -- if bypass(token_else)
      if (BYPASS(TOKEN_END)) then
        if (BYPASS(TOKEN_IF)) then
          if (BYPASS(TOKEN_SEMICOLON)) then
            NESTING_METRIC(IF_END);
            return (TRUE);
          else
            SYNTAX_ERROR("If statement");
          end if;                    -- if bypass(token_semicolon)
        else
          SYNTAX_ERROR("If statement");
        end if;                      -- if bypass(token_if)
      else
        SYNTAX_ERROR("If statement");
      end if;                        -- if bypass(token_end)
    else
      SYNTAX_ERROR("If statement");
    end if;                          -- if sequence_of_statements
  else
    SYNTAX_ERROR("If statement");
  end if;                            -- if bypass(token_then)
 else
  return (FALSE);
 end if;                             -- if expression statement
end IF_STATEMENT;

----------------------------------------------------------------------------

-- CASE_STATEMENT -- : EXPRESSION is CASE_STATEMENT_ALTERNATIVE : end case ;
function CASE_STATEMENT return boolean is
begin
put(RESULT_FILE, "In case statement "); new_line(RESULT_FILE);
 if (EXPRESSION) then
  if (BYPASS(TOKEN_IS)) then
   if (CASE_STATEMENT_ALTERNATIVE) then
    while (CASE_STATEMENT_ALTERNATIVE) loop
      null;
    end loop;
    if (BYPASS(TOKEN_END)) then
     if (BYPASS(TOKEN_CASE)) then
      if (BYPASS(TOKEN_SEMICOLON)) then
        NESTING_METRIC(CASE_END);
        return (TRUE);
      else
```

```
        SYNTAX_ERROR("Case statement");
      end if;                    -- if bypass(token_semicolon)
    else
      SYNTAX_ERROR("Case statement");
    end if;                      -- if bypass(token_case)
   else
    SYNTAX_ERROR("Case statement");
   end if;                       -- if bypass(token_end)
  else
   SYNTAX_ERROR("Case statement");
  end if;                        -- if case_statement_alternative
 else
  SYNTAX_ERROR("Case statement");
 end if;                         -- if bypass(token_is)
else
 return (FALSE);
end if;                          -- if expression statement
end CASE_STATEMENT;

------------------------------------------------------------------------------

-- CASE_STATEMENT_ALTERNATIVE --> when CHOICE {| CHOICE}* =>
--                                   SEQUENCE_OF_STATEMENTS
function CASE_STATEMENT_ALTERNATIVE return boolean is
begin
put(RESULT_FILE, "In case_statement_alternative "); new_line(RESULT_FILE);
 if (BYPASS(TOKEN_WHEN)) then
  if (CHOICE) then
   while (BYPASS(TOKEN_BAR)) loop
    if not (CHOICE) then
     SYNTAX_ERROR("Case statement alternative");
    end if;                      -- if not choice statement
   end loop;
   if (BYPASS(TOKEN_ARROW)) then
    if (SEQUENCE_OF_STATEMENTS) then
     return (TRUE);
    else
     SYNTAX_ERROR("Case statement alternative");
    end if;                      -- if sequence_of_statements
   else
    SYNTAX_ERROR("Case statement alternative");
   end if;                       -- if bypass(token_arrow)
  else
   SYNTAX_ERROR("Case statement alternative");
  end if;                        -- if choice statement
 else
  return (FALSE);
 end if;                         -- if bypass(token_when)
end CASE_STATEMENT_ALTERNATIVE;

------------------------------------------------------------------------------
```

```
 -- LOOP_STATEMENT --> [identifier : ]? [ITERATION_SCHEME ]? loop
 --                     SEQUENCE_OF_STATEMENTS end loop [identifier ]? ;
function LOOP_STATEMENT return boolean is
begin
put(RESULT_FILE, "In loop_statement "); new_line(RESULT_FILE);
 if (BYPASS(TOKEN_IDENTIFIER)) then
  if (BYPASS(TOKEN_COLON)) then
   null;
  else
   SYNTAX_ERROR("Loop statement");
  end if;                    -- if bypass(token_colon)
 end if;                     -- if bypass(token_identifier)
 if (ITERATION_SCHEME) then
  NO_ITERATION := FALSE;
 end if;                     -- if iteration_scheme statement
 if (BYPASS(TOKEN_LOOP)) then
  if (NO_ITERATION) then
   NESTING_METRIC(LOOP_CONSTRUCT);
  else
   NO_ITERATION := TRUE;
  end if;
  if (SEQUENCE_OF_STATEMENTS) then
   if (BYPASS(TOKEN_END)) then
    if (BYPASS(TOKEN_LOOP)) then
     if (BYPASS(TOKEN_IDENTIFIER)) then
      null;
     end if;                 -- if bypass(token_identifier)
     if (BYPASS(TOKEN_SEMICOLON)) then
      NESTING_METRIC(LOOP_END);
      return (TRUE);
     else
      SYNTAX_ERROR("Loop statement");
     end if;                 -- if bypass(token_semicolon)
    else
     SYNTAX_ERROR("Loop statement");
    end if;                  -- if bypass(token_loop)
   else
    SYNTAX_ERROR("Loop statement");
   end if;                   -- if bypass(token_end)
  else
   SYNTAX_ERROR("Loop statement");
  end if;                    -- if sequence_of_statements
 else
  return (FALSE);
 end if;                     -- if bypass(token_loop)
end LOOP_STATEMENT;

---------------------------------------------------------------------

 -- EXCEPTION_HANDLER --> when EXCEPTION_CHOICE [| EXCEPTION_CHOICE]* =>
 --                        SEQUENCE_OF_STATEMENTS
function EXCEPTION_HANDLER return boolean is
```

```
begin
put(RESULT_FILE, "In exception_handler "); new_line(RESULT_FILE);
 if (BYPASS(TOKEN_WHEN)) then
  if (EXCEPTION_CHOICE) then
   while (BYPASS(TOKEN_BAR)) loop
    if not (EXCEPTION_CHOICE) then
     SYNTAX_ERROR("Exception handler");
    end if;                          -- if not exception_choice
   end loop;
   if (BYPASS(TOKEN_ARROW)) then
    if (SEQUENCE_OF_STATEMENTS) then
     return (TRUE);
    else
     SYNTAX_ERROR("Exception handler");
    end if;                          -- if sequence_of_statements
   else
    SYNTAX_ERROR("Exception handler");
   end if;                           -- if bypass(token_arrow)
  else
   SYNTAX_ERROR("Exception handler");
  end if;                            -- if exception_choice statement
 else
  return (FALSE);
 end if;                       -- if bypass(token-when)
end EXCEPTION_HANDLER;

-----------------------------------------------------------------------------

 -- ACCEPT_STATEMENT --> identifier [(EXPRESSION) ?] [FORMAL_PART ?
 --                      do SEQUENCE_OF_STATEMENTS end [identifier ? ] ? ] ;
function ACCEPT_STATEMENT return boolean is
begin
put(RESULT_FILE, "In accept statement "); new_line(RESULT_FILE);
 if (BYPASS(TOKEN_IDENTIFIER)) then
  if (BYPASS(TOKEN_LEFT_PAREN)) then
   if (EXPRESSION) then
    if (BYPASS(TOKEN_RIGHT_PAREN)) then
     null;
    else
     SYNTAX_ERROR("Accept statement");
    end if;                          -- if bypass(token_right_paren)
   else
    SYNTAX_ERROR("Accept statement");
   end if;                           -- if expression statement
  end if;                            -- if bypass(token_left_paren)
  if (FORMAL_PART) then
   null;
  end if;                            -- if formal_part statement
  if (BYPASS(TOKEN_DO)) then
   if (SEQUENCE_OF_STATEMENTS) then
    if (BYPASS(TOKEN_END)) then
     if (BYPASS(TOKEN_IDENTIFIER)) then
```

```
        null;
       end if;                    -- if bypass(token_identifier)
      else
       SYNTAX_ERROR("Accept statement");
      end if;                     -- if bypass(token_end)
     else
      SYNTAX_ERROR("Accept statement");
     end if;                          -- if sequence_of_statements
    end if;                           -- if bypass(token_do)
    if (BYPASS(TOKEN_SEMICOLON)) then
     return (TRUE);
    else
     SYNTAX_ERROR("Accept statement");
    end if;                           -- if bypass(token_semicolon)
   else
    return (FALSE);
   end if;                    -- if bypass(token_identifier)
end ACCEPT_STATEMENT;

--------------------------------------------------------------------------

 -- SELECT_STATEMENT --> SELECT_STATEMENT_TAIL SELECT_ENTRY_CALL end select ;
function SELECT_STATEMENT return boolean is
begin
put(RESULT_FILE, "In select_statement "); new_line(RESULT_FILE);
 if (SELECT_STATEMENT_TAIL) then
  if (SELECT_ENTRY_CALL) then
   if (BYPASS(TOKEN_END)) then
    if (BYPASS(TOKEN_SELECT)) then
     if (BYPASS(TOKEN_SEMICOLON)) then
      return (TRUE);
     else
      SYNTAX_ERROR("Select statement");
     end if;                  -- if bypass(token_semicolon)
    else
     SYNTAX_ERROR("Select statement");
    end if;                           -- if bypass(token_select)
   else
    SYNTAX_ERROR("Select statement");
   end if;                            -- if bypass(token_end)
  else
   SYNTAX_ERROR("Select statement");
  end if;                             -- if select_entry_call statement
 else
  return (FALSE);
 end if;                      -- if select_statement_tail
end SELECT_STATEMENT;

--------------------------------------------------------------------------

 -- SELECT_STATEMENT_TAIL --> SELECT_ALTERNATIVE or SELECT_ALTERNATIVE *
 --                        --> NAME : SEQUENCE_OF_STATEMENTS ?
```

```
function SELECT_STATEMENT_TAIL return boolean is
begin
put(RESULT_FILE, "In select_statement_tail "); new_line(RESULT_FILE);
 if (SELECT_ALTERNATIVE) then
  while (BYPASS(TOKEN_OR)) loop
   if not (SELECT_ALTERNATIVE) then
    SYNTAX_ERROR("Select statement tail");
   end if;
  end loop;
  return (TRUE);
 elsif (NAME) then                    -- check for entry call statement
  if (BYPASS(TOKEN_SEMICOLON)) then
   if (SEQUENCE_OF_STATEMENTS) then
    null;
   end if;                            -- if sequence_of_statements
   return (TRUE);
  else
   SYNTAX_ERROR("Select statement tail");
  end if;                             -- if bypass(token_semicolon)
 else
  return (FALSE);
 end if;                       -- if select_alternative statement
end SELECT_STATEMENT_TAIL;

-------------------------------------------------------------------------------

 -- SELECT_ALTERNATIVE --> [when EXPRESSION => ?] accept ACCEPT_STATEMENT
 --                        [SEQUENCE_OF_STATEMENTS ?]
 --                    --> [when EXPRESSION => ?] delay DELAY_STATEMENT
 --                        [SEQUENCE_OF_STATEMENTS ?]
 --                    --> [when EXPRESSION => ?] terminate ;
function SELECT_ALTERNATIVE return boolean is
begin
put(RESULT_FILE, "In select_alternative "); new_line(RESULT_FILE);
 if (BYPASS(TOKEN_WHEN)) then
  if (EXPRESSION) then
   if (BYPASS(TOKEN_ARROW)) then
    null;
   else
    SYNTAX_ERROR("Select alternative");
   end if;                            -- if bypass(token_arrow)
  else
   SYNTAX_ERROR("Select alternative");
  end if;                             -- if expression statement
 end if;                       -- if bypass(token_when)
 if (BYPASS(TOKEN_ACCEPT)) then
  if (ACCEPT_STATEMENT) then
   if (SEQUENCE_OF_STATEMENTS) then
    null;
   end if;                            -- if sequence_of_statements
   return (TRUE);
  else
```

```
    SYNTAX_ERROR("Select alternative");
   end if;                              -- if accept_statement
  elsif (BYPASS(TOKEN_DELAY)) then
   if (DELAY_STATEMENT) then
    if (SEQUENCE_OF_STATEMENTS) then
     null;
    end if;                             -- if sequence_of_statements
    return (TRUE);
   else
    SYNTAX_ERROR("Select alternative");
   end if;                              -- if delay_statement
  elsif (BYPASS(TOKEN_TERMINATE)) then
   if (BYPASS(TOKEN_SEMICOLON)) then
    return (TRUE);
   else
    SYNTAX_ERROR("Select alternative");
   end if;                              -- if bypass(token_semicolon)
  else
   return (FALSE);
  end if;                         -- if bypass(token_accept)
end SELECT_ALTERNATIVE;

-----------------------------------------------------------------------------

 -- SELECT_ENTRY_CALL --> else SEQUENCE_OF_STATEMENTS
 --                   --> or delay DELAY_STATEMENT [SEQUENCE_OF_STATEMENTS ?
function SELECT_ENTRY_CALL return boolean is
begin
put(RESULT_FILE, "In select_entry_call "); new_line(RESULT_FILE);
 if (BYPASS(TOKEN_ELSE)) then
  if (SEQUENCE_OF_STATEMENTS) then
   return (TRUE);
  else
   SYNTAX_ERROR("Select entry call");
  end if;                              -- if sequence_of_statements
 elsif (BYPASS(TOKEN_OR)) then
  if (BYPASS(TOKEN_DELAY)) then
   if (DELAY_STATEMENT) then
    if (SEQUENCE_OF_STATEMENTS) then
     null;
    end if;                            -- if sequence_of_statements
    return (TRUE);
   else
    SYNTAX_ERROR("Select entry call");
   end if;                             -- if delay_statement
  else
   SYNTAX_ERROR("Select entry call");
  end if;                              -- if bypass(token_delay)
 else
  return (FALSE);
 end if;                         -- if bypass(token_else)
end SELECT_ENTRY_CALL;
```

```
end PARSER_1;


--*************************************************************--
--
-- TITLE:        AN ADA SOFTWARE METRIC
--
-- MODULE NAME:   PACKAGE PARSER_2
-- DATE CREATED:  18 JUL 86
-- LAST MODIFIED: 30 MAY 87
--
-- AUTHORS:       LCDR JEFFREY L. NIEDER
--                LT KARL S. FAIRBANKS, JR.
--                LCDR PAUL M. HERZIG
-- DESCRIPTION: This package contains thirty-three functions
--     that are the middle level productions for our top-down,
--     recursive descent parser.  Each function is preceded
--     by the grammaar productions they are implementing.
--
--*************************************************************--

with PARSER_3, PARSER_4, HENRY_GLOBAL, HENRY, BYPASS_FUNCTION,
   BYPASS_SUPPORT_FUNCTIONS, GLOBAL_PARSER, GLOBAL, TEXT_IO;
use PARSER_3, PARSER_4, HENRY_GLOBAL, HENRY, BYPASS_FUNCTION,
   BYPASS_SUPPORT_FUNCTIONS, GLOBAL_PARSER, GLOBAL, TEXT_IO;

package PARSER_2 is
 IDENT_DECLARE : BOOLEAN := FALSE;
 function GENERIC_ACTUAL_PART return boolean;
 function GENERIC_ASSOCIATION return boolean;
 function GENERIC_FORMAL_PARAMETER return boolean;
 function GENERIC_TYPE_DEFINITION return boolean;
 function PRIVATE_TYPE_DECLARATION return boolean;
 function TYPE_DECLARATION return boolean;
 function SUBTYPE_DECLARATION return boolean;
 function DISCRIMINANT_PART return boolean;
 function DISCRIMINANT_SPECIFICATION return boolean;
 function TYPE_DEFINITION return boolean;
 function RECORD_TYPE_DEFINITION return boolean;
 function COMPONENT_LIST return boolean;
 function COMPONENT_DECLARATION return boolean;
 function VARIANT_PART return boolean;
 function VARIANT return boolean;
 function WITH_OR_USE_CLAUSE return boolean;
 function FORMAL_PART return boolean;
 function IDENTIFIER_DECLARATION return boolean;
 function IDENTIFIER_DECLARATION_TAIL return boolean;
 function EXCEPTION_TAIL return boolean;
 function EXCEPTION_CHOICE return boolean;
 function CONSTANT_TERM return boolean;
```

```
  function IDENTIFIER_TAIL return boolean;
  function PARAMETER_SPECIFICATION return boolean;
  function IDENTIFIER_LIST return boolean;
  function MODE return boolean;
  function DESIGNATOR return boolean;
  function SIMPLE_STATEMENT return boolean;
  function ASSIGNMENT_OR_PROCEDURE_CALL return boolean;
  function LABEL return boolean;
  function ENTRY_DECLARATION return boolean;
  function REPRESENTATION_CLAUSE return boolean;
  function RECORD_REPRESENTATION_CLAUSE return boolean;
end PARSER_2;

-------------------------------------------------------------------------
-------------------------------------------------------------------------

package body PARSER_2 is

  -- GENERIC_ACTUAL_PART --> (GENERIC_ASSOCIATION {, GENERIC_ASSOCIATION} )
function GENERIC_ACTUAL_PART return boolean is
begin
  if (BYPASS(TOKEN_LEFT_PAREN)) then
    if (GENERIC_ASSOCIATION) then
      while (BYPASS(TOKEN_COMMA)) loop
        if not (GENERIC_ASSOCIATION) then
          SYNTAX_ERROR("Generic actual part");
        end if;                          -- if not generic_association
      end loop;
      if (BYPASS(TOKEN_RIGHT_PAREN)) then
        return (TRUE);
      else
        SYNTAX_ERROR("Generic actual part");
      end if;                            -- if bypass(token_right_paren)
    else
      SYNTAX_ERROR("Generic actual part");
    end if;                              -- if generic association statement
  else
    return(FALSE);
  end if;                        -- if bypass(token_left_paren)
end GENERIC_ACTUAL_PART;

-------------------------------------------------------------------------

  -- GENERIC_ASSOCIATION --> [GENERIC_FORMAL_PARAMETER ?] EXPRESSION
function GENERIC_ASSOCIATION return boolean is
begin
  if (GENERIC_FORMAL_PARAMETER) then
    null;
  end if;                        -- if generic_formal_parameter statement
  if (EXPRESSION) then                          -- check for generic_actual_parameter
    return (TRUE);
  else
```

```
    return (FALSE);
  end if;                        -- if expression
end GENERIC_ASSOCIATION;

------------------------------------------------------------------------

  -- GENERIC_FORMAL_PARAMETER --> identifier =>
  --                           --> string_literal =>
function GENERIC_FORMAL_PARAMETER return boolean is
begin
  LOOK_AHEAD_TOKEN := TOKEN_RECORD_BUFFER(TOKEN_ARRAY_INDEX + 1);
  if (ADJUST_LEXEME(LOOK_AHEAD_TOKEN.LEXEME,
               LOOK_AHEAD_TOKEN.LEXEME_SIZE - 1) = "=>") then
    if (BYPASS(TOKEN_IDENTIFIER)) then
      if (BYPASS(TOKEN_ARROW)) then
        return (TRUE);
      else
        SYNTAX_ERROR("Generic formal parameter");
      end if;                       -- if bypass(token_arrow)
    elsif (BYPASS(TOKEN_STRING_LITERAL)) then
      if (BYPASS(TOKEN_ARROW)) then
        return (TRUE);
      else
        SYNTAX_ERROR("Generic formal parameter");
      end if;                       -- if bypass(token_arrow)
    else
      SYNTAX_ERROR("Generic formal parameter");
    end if;                         -- if bypass(token_identifier)
  else
    return (FALSE);
  end if;                        -- if adjust_lexeme(lookahead_token)
end GENERIC_FORMAL_PARAMETER;

------------------------------------------------------------------------

  -- GENERIC_TYPE_DEFINITION --> ( <> )
  --                          --> range <>
  --                          --> digits <>
  --                          --> delta <>
  --                          --> array ARRAY_TYPE_DEFINITION
  --                          --> access SUBTYPE_INDICATION
function GENERIC_TYPE_DEFINITION return boolean is
begin
  if (BYPASS(TOKEN_LEFT_PAREN)) then
    if (BYPASS(TOKEN_BRACKETS)) then
      if (BYPASS(TOKEN_RIGHT_PAREN)) then
        return (TRUE);
      else
        SYNTAX_ERROR("Generic type definition");
      end if;                       -- if bypass(token_right_paren)
    else
      SYNTAX_ERROR("Generic type definition");
```

```
    end if;                                  -- if bypass(token_brackets)
  elsif (BYPASS(TOKEN_RANGE)) or else (BYPASS(TOKEN_DIGITS))
    or else (BYPASS(TOKEN_DELTA)) then
    if (BYPASS(TOKEN_BRACKETS)) then
      return (TRUE);
    else
      SYNTAX_ERROR("Generic type definition");
    end if;                                  -- if bypass(token_brackets)
  elsif (BYPASS(TOKEN_ARRAY)) then
    if (ARRAY_TYPE_DEFINITION) then
      return (TRUE);
    else
      SYNTAX_ERROR("Generic type definition");
    end if;                                  -- if array_type_definition
  elsif (BYPASS(TOKEN_ACCESS)) then
    if (SUBTYPE_INDICATION) then
      return (TRUE);
    else
      SYNTAX_ERROR("Generic type definition");
    end if;                                  -- if subtype_indication
  else
    return (FALSE);
  end if;                          -- if bypass(token_left_paren)
end GENERIC_TYPE_DEFINITION;

------------------------------------------------------------------------

  -- PRIVATE_TYPE_DECLARATION --> [limited ?] private
function PRIVATE_TYPE_DECLARATION return boolean is
begin
  if (BYPASS(TOKEN_LIMITED)) then
    null;
  end if;
  if (BYPASS(TOKEN_PRIVATE)) then
    return (TRUE);
  else
    return (FALSE);
  end if;
end PRIVATE_TYPE_DECLARATION;

------------------------------------------------------------------------

  -- SUBTYPE_DECLARATION --> identifier is SUBTYPE_INDICATION ;
function SUBTYPE_DECLARATION return boolean is
begin
  if (BYPASS(TOKEN_IDENTIFIER)) then
    if (BYPASS(TOKEN_IS)) then
      if (SUBTYPE_INDICATION) then
        if (BYPASS(TOKEN_SEMICOLON)) then
          return (TRUE);
        else
          SYNTAX_ERROR("Subtype declaration");
```

```
      end if;                              -- if bypass(token_semicolon)
     else
      SYNTAX_ERROR("Subtype declaration");
     end if;                    -- if subtype_indication statement
    else
     SYNTAX_ERROR("Subtype declaration");
    end if;                     -- if bypass(token_is)
   else
    return (FALSE);
   end if;                      -- if bypass(token_identifier)
  end SUBTYPE_DECLARATION;

  ---------------------------------------------------------------------------

   -- TYPE_DECLARATION --> identifier DISCRIMINANT_PART ?
   --                        is SUBTYPE_INDICATION;
  function TYPE_DECLARATION return boolean is
  begin
   if (BYPASS(TOKEN_IDENTIFIER)) then
    if (DISCRIMINANT_PART) then
     null;
    end if;                     -- if discriminant_part statement
    if (BYPASS(TOKEN_IS)) then        -- declaration is full_type if 'is'
     if (PRIVATE_TYPE_DECLARATION) then
      null;
     elsif (TYPE_DEFINITION) then    --    present, otherwise incomplete_type
      null;
     else
      SYNTAX_ERROR("Type declaration");
     end if;                    -- if type_definition statement
    end if;                     -- if bypass(token_is)
    if (BYPASS(TOKEN_SEMICOLON)) then
     return (TRUE);
    else
     SYNTAX_ERROR("Type declaration");
    end if;                     -- if bypass(token_semicolon)
   else
    return (FALSE);
   end if;                      -- if bypass(token_identifier)
  end TYPE_DECLARATION;

  ---------------------------------------------------------------------------

   -- DISCRIMINANT_PART --> (DISCRIMINANT_SPECIFICATION
   --                          ; DISCRIMINANT_SPECIFICATION * )
  function DISCRIMINANT_PART return boolean is
  begin
   if (BYPASS(TOKEN_LEFT_PAREN)) then
    if (DISCRIMINANT_SPECIFICATION) then
     while (BYPASS(TOKEN_SEMICOLON)) loop
      if not (DISCRIMINANT_SPECIFICATION) then
       SYNTAX_ERROR("Discriminant part");
```

```
      end if;                  -- if not discriminant_specification
    end loop;
    if (BYPASS(TOKEN_RIGHT_PAREN)) then
      return (TRUE);
    else
      SYNTAX_ERROR("Discriminant part");
    end if;                    -- if bypass(token_right_paren)
  else
    SYNTAX_ERROR("Discriminant part");
  end if;                      -- if discriminant_specification
else
  return (FALSE);
end if;                        -- if bypass(token_left_paren)
end DISCRIMINANT_PART;

----------------------------------------------------------------------------

  -- DISCRIMINANT_SPECIFICATION --> IDENTIFIER_LIST : NAME := EXPRESSION ?
function DISCRIMINANT_SPECIFICATION return boolean is
begin
  if (IDENTIFIER_LIST) then
    if (BYPASS(TOKEN_COLON)) then
      if (NAME) then                        -- check for type_mark
        if (BYPASS(TOKEN_ASSIGNMENT)) then
          if (EXPRESSION) then
            null;
          else
            SYNTAX_ERROR("Discriminant specification");
          end if;                   -- if expression statement
        end if;                     -- if bypass(token_assignment)
        return (TRUE);
      else
        SYNTAX_ERROR("Discriminant specification");
      end if;                       -- if name statement
    else
      SYNTAX_ERROR("Discriminant specification");
    end if;                         -- if bypass(token_colon)
  else
    return (FALSE);
  end if;                           -- if identifier_list statement
end DISCRIMINANT_SPECIFICATION;

----------------------------------------------------------------------------

  -- TYPE_DEFINITION --> ENUMERATION_TYPE_DEFINITION
  --                 --> INTEGER_TYPE_DEFINITION
  --                 --> digits FLOATING_OR_FIXED_POINT_CONSTRAINT
  --                 --> delta FLOATING_OR_FIXED_POINT_CONSTRAINT
  --                 --> array ARRAY_TYPE_DEFINITION
  --                 --> record RECORD_TYPE_DEFINITION
  --                 --> access SUBTYPE_INDICATION
  --                 --> new SUBTYPE_INDICATION
```

```
function TYPE_DEFINITION return boolean is
begin
 if (ENUMERATION_TYPE_DEFINITION) then
  return (TRUE);
 elsif (INTEGER_TYPE_DEFINITION) then
  return (TRUE);
 elsif (BYPASS(TOKEN_DIGITS)) or else (BYPASS(TOKEN_DELTA)) then
  if (FLOATING_OR_FIXED_POINT_CONSTRAINT) then
   return (TRUE);
  else
   SYNTAX_ERROR("Type definition");
  end if;                          -- floating_or_fixed_point_constraint
 elsif (BYPASS(TOKEN_ARRAY)) then
  if (ARRAY_TYPE_DEFINITION) then
   return (TRUE);
  else
   SYNTAX_ERROR("Type definition");
  end if;                          -- if array_type_definition
 elsif (BYPASS(TOKEN_RECORD_STRUCTURE)) then
  if (RECORD_TYPE_DEFINITION) then
   return (TRUE);
  else
   SYNTAX_ERROR("Type definition");
  end if;                          -- if record_type_definition
 elsif (BYPASS(TOKEN_ACCESS)) or else (BYPASS(TOKEN_NEW)) then
  if (SUBTYPE_INDICATION) then
   return (TRUE);
  else
   SYNTAX_ERROR("Type definition");
  end if;                          -- if subtype_indication
 else
  return (FALSE);
 end if;
end TYPE_DEFINITION;

-------------------------------------------------------------------------

 -- RECORD_TYPE_DEFINITION --> COMPONENT_LIST end record
function RECORD_TYPE_DEFINITION return boolean is
begin
 if (COMPONENT_LIST) then
  if (BYPASS(TOKEN_END)) then
   if (BYPASS(TOKEN_RECORD_STRUCTURE)) then
    return (TRUE);
   else
    SYNTAX_ERROR("Record type definition");
   end if;                         -- if bypass(token_record-structure)
  else
   SYNTAX_ERROR("Record type definition");
  end if;                          -- if bypass(token_end)
 else
  return (FALSE);
```

```
  end if;                         -- if component_list statement
end RECORD_TYPE_DEFINITION;

---------------------------------------------------------------

  -- COMPONENT_LIST --> COMPONENT_DECLARATION * VAARIANT_PART ?
  --                --> null ;
function COMPONENT_LIST return boolean is
begin
  while (COMPONENT_DECLARATION) loop
    null;
  end loop;
  if (VARIANT_PART) then
    null;
  elsif (BYPASS(TOKEN_NULL)) then
    if (BYPASS(TOKEN_SEMICOLON)) then
      null;
    end if;
  end if;
  return (TRUE);
end COMPONENT_LIST;

---------------------------------------------------------------

  -- COMPONENT_DECLARATION --> IDENTIFIER_LIST : SUBTYPE_INDICATION
  --                            := EXPRESSION ? ;
function COMPONENT_DECLARATION return boolean is
begin
  if (IDENTIFIER_LIST) then
    if (BYPASS(TOKEN_COLON)) then
      if (SUBTYPE_INDICATION) then
        if (BYPASS(TOKEN_ASSIGNMENT)) then
          if (EXPRESSION) then
            if (BYPASS(TOKEN_SEMICOLON)) then
              return (TRUE);
            else
              SYNTAX_ERROR("Component declaration");
            end if;              -- if bypass(token_semicolon)
          else
            SYNTAX_ERROR("Component declaration");
          end if;                -- if expression statement
        end if;                        -- if bypass(token_assignment)
        if (BYPASS(TOKEN_SEMICOLON)) then
          return (TRUE);
        else
          SYNTAX_ERROR("Component declaration");
        end if;                        -- if bypass(token_semicolon)
      else
        SYNTAX_ERROR("Component declaration");
      end if;                          -- if subtype_indication statement
    else
      SYNTAX_ERROR("Component declaration");
```

```
    end if;                              -- if bypass(token_colon)
  else
    return (FALSE);
  end if;                          -- if identifier_list statement
end COMPONENT_DECLARATION;

----------------------------------------------------------------------------

  -- VARIANT_PART --> case identifier is {VARIANT} end case ;
function VARIANT_PART return boolean is
begin
  if (BYPASS(TOKEN_CASE)) then
    if (BYPASS(TOKEN_IDENTIFIER)) then
      if (BYPASS(TOKEN_IS)) then
        if (VARIANT) then
          while (VARIANT) loop
            null;
          end loop;
          if (BYPASS(TOKEN_END)) then
            if (BYPASS(TOKEN_CASE)) then
              if (BYPASS(TOKEN_SEMICOLON)) then
                return (TRUE);
              else
                SYNTAX_ERROR("Variant part");
              end if;              -- if bypass(token_semicolon)
            else
              SYNTAX_ERROR("Variant part");
            end if;                -- if bypass(token_case)
          else
            SYNTAX_ERROR("Variant part");
          end if;                  -- if bypass(token_end)
        else
          SYNTAX_ERROR("Variant part");
        end if;                            -- if variant statement
      else
        SYNTAX_ERROR("Variant part");
      end if;                              -- if bypass(token_is)
    else
      SYNTAX_ERROR("Variant part");
    end if;                                -- if bypass(token_identifier)
  else
    return (FALSE);
  end if;                          -- if bypass(token_case)
end VARIANT_PART;

----------------------------------------------------------------------------

  -- VARIANT --> when CHOICE { | CHOICE } => COMPONENT_LIST
function VARIANT return boolean is
begin
  if (BYPASS(TOKEN_WHEN)) then
    if (CHOICE) then
```

```
    while (BYPASS(TOKEN_BAR)) loop
      if not (CHOICE) then
        SYNTAX_ERROR("Variant");
      end if;                          -- if not choice statement
    end loop;
    if (BYPASS(TOKEN_ARROW)) then
      if (COMPONENT_LIST) then
        return (TRUE);
      else
        SYNTAX_ERROR("Variant");
      end if;                          -- if component_list statement
    else
      SYNTAX_ERROR("Variant");
    end if;                            -- if bypass(token_arrow)
  else
    SYNTAX_ERROR("Variant");
  end if;                              -- if choice statement
 else
  return (FALSE);
 end if;                       -- if bypass(token_when)
end VARIANT;

----------------------------------------------------------------------------

 -- WITH_OR_USE_CLAUSE --> identifier {, identifier * ;
function WITH_OR_USE_CLAUSE return boolean is
begin
 if (BYPASS(TOKEN_IDENTIFIER)) then
   while (BYPASS(TOKEN_COMMA)) loop
     if not (BYPASS(TOKEN_IDENTIFIER)) then
       SYNTAX_ERROR("With or use clause");
     end if;
   end loop;
   if (BYPASS(TOKEN_SEMICOLON)) then
     return (TRUE);
   else
     SYNTAX_ERROR("With or use clause");
   end if;                     -- if bypass(token_semicolon)
 else
   return (FALSE);
 end if;                       -- if bypass(token_identifier)
end WITH_OR_USE_CLAUSE;

----------------------------------------------------------------------------

 -- FORMAL_PART --> (PARAMETER_SPECIFICATION ; PARAMETER_SPECIFICATION * )
function FORMAL_PART return boolean is
begin
 if (BYPASS(TOKEN_LEFT_PAREN)) then
    FORMAL_PARAM_DECLARE := TRUE;
    if (PARAMETER_SPECIFICATION) then
    while (BYPASS(TOKEN_SEMICOLON)) loop
```

```
      if not (PARAMETER_SPECIFICATION) then
        SYNTAX_ERROR("Formal part");
      end if;             -- if not parameter_specification statement
    end loop;
    if (BYPASS(TOKEN_RIGHT_PAREN)) then
      if PACKAGE_BODY_DECLARE then
      WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, END_PARAM_DECLARE,
              NONE, NEXT_HEN);
      CREATE_NODE(NEXT_HEN, LAST_RECORD);
      end if;
      FORMAL_PARAM_DECLARE := FALSE;
      return (TRUE);
    else
      SYNTAX_ERROR("Formal part");
    end if;             -- if bypass(token_right_paren) statement
  else
    SYNTAX_ERROR("Formal part");
  end if;               -- if parameter_specification statement
else
  return (FALSE);
end if;                 -- if bypass(token_left_paren) statement
end FORMAL_PART;

--------------------------------------------------------------------------

-- IDENTIFIER_DECLARATION --> IDENTIFIER_LIST : IDENTIFIER_DECLARATION_TAIL
function IDENTIFIER_DECLARATION return boolean is
begin
put(RESULT_FILE, "IN IDENTIFIER DECLARATION"); NEW_LINE(RESULT_FILE);
 HENRY_WRITE_ENABLE := TRUE;
 IDENT_DECLARE := TRUE;
 if (IDENTIFIER_LIST) then
  if (BYPASS(TOKEN_COLON)) then
    if (IDENTIFIER_DECLARATION_TAIL) then
      HENRY_WRITE_ENABLE := FALSE;

      return (TRUE);
    else
      SYNTAX_ERROR("Identifier declaration");
    end if;             -- if identifier list statement
  else
    SYNTAX_ERROR("Identifier declaration");
  end if;               -- if bypass(token_colon)
 else
  return(FALSE);
 end if                 -- if identifier list statement
end IDENTIFIER_DECLARATION;

--------------------------------------------------------------------------

-- IDENTIFIER_DECLARATION_TAIL --> exception EXCEPTION_TAIL
--                             --> constant CONSTANT_TERM
```

```
--                                  --> array ARRAY TYPE DEFINITION
--                                          := EXPRESSION ? ;
--                                  --> NAME IDENTIFIER TAIL
function IDENTIFIER_DECLARATION_TAIL return boolean is
begin
put(RESULT_FILE, "IN IDENTIFIER DECLARATION TAIL"); NEW_LINE(RESULT_FILE);
 if (BYPASS(TOKEN_EXCEPTION)) then
  if (EXCEPTION_TAIL) then
   return (TRUE);
  else
   SYNTAX_ERROR("Identifier declaration tail");
  end if;                   -- if exception tail statement
 elsif (BYPASS(TOKEN_CONSTANT)) then
  if (CONSTANT_TERM) then
   return (TRUE);
  else
   SYNTAX_ERROR("Identifier declaration tail");
  end if;                              -- if constant term statement
 elsif (BYPASS(TOKEN_ARRAY)) then
  if (ARRAY_TYPE_DEFINITION) then
   if (BYPASS(TOKEN_ASSIGNMENT)) then
    if (EXPRESSION) then
     null;
    else
     SYNTAX_ERROR("Identifier declaration tail");
    end if;                            -- if expression statement
   end if;                             -- if bypass(token_assignment)
  else
   SYNTAX_ERROR("Identifier declaration tail");
  end if;                              -- if array_type_definition
  if (BYPASS(TOKEN_SEMICOLON)) then
   return (TRUE);
  else
   SYNTAX_ERROR("Identifier declaration tail");
  end if;                              -- if bypass(token_semicolon)
 elsif (NAME) then
  if (IDENTIFIER_TAIL) then
   return (TRUE);
  else
   SYNTAX_ERROR("Identifier declaration tail");
  end if;                              -- if identifier tail
 else
  return (FALSE);
 end if;                         -- if bypass(token_exception)
end IDENTIFIER_DECLARATION_TAIL;

---------------------------------------------------------------------------

 -- EXCEPTION_TAIL --> ;
 --              --> renames NAME ;
function EXCEPTION_TAIL return boolean is
begin
```

```
  if (BYPASS(TOKEN_SEMICOLON)) then
    return (TRUE);
  elsif (BYPASS(TOKEN_RENAMES)) then
    if (NAME) then
      if (BYPASS(TOKEN_SEMICOLON)) then
        return (TRUE);
      else
        SYNTAX_ERROR("Exception tail");
      end if;                    -- if bypass(token_semicolon)
    else
      SYNTAX_ERROR("Exception tail");
    end if;                      -- if name statement
  else
    return (FALSE);
  end if;                        -- if bypass(token_semicolon)
end EXCEPTION_TAIL;

--------------------------------------------------------------------------

  -- EXCEPTION_CHOICE --> identifier
  --                  --> others
function EXCEPTION_CHOICE return boolean is
begin
  if (BYPASS(TOKEN_IDENTIFIER)) then
    return (TRUE);
  elsif (BYPASS(TOKEN_OTHERS)) then
    return (TRUE);
  else
    return (FALSE);
  end if;
end EXCEPTION_CHOICE;

--------------------------------------------------------------------------

  -- CONSTANT_TERM --> array ARRAY_TYPE_DEFINITION := EXPRESSION ? ;
  --               --> := EXPRESSION ;
  --               --> NAME IDENTIFIER_TAIL
function CONSTANT_TERM return boolean is
begin
  if (BYPASS(TOKEN_ARRAY)) then
    if (ARRAY_TYPE_DEFINITION) then
      if (BYPASS(TOKEN_ASSIGNMENT)) then
        if (EXPRESSION) then
          null;
        else
          SYNTAX_ERROR("Constant term");
        end if;                  -- if expression statement
      end if;                    -- if bypass(token_assignment)
    else
      SYNTAX_ERROR("Constant term");
    end if;                      -- if array_type_definition
    if (BYPASS(TOKEN_SEMICOLON)) then
```

```
      return (TRUE);
    else
      SYNTAX_ERROR("Constant term");
    end if;                            -- if bypass(token_semicolon)
  elsif (BYPASS(TOKEN_ASSIGNMENT)) then
    if (EXPRESSION) then
      if (BYPASS(TOKEN_SEMICOLON)) then
        return (TRUE);
      else
        SYNTAX_ERROR("Constant term");
      end if;                          -- if bypass(token_semicolon)
    else
      SYNTAX_ERROR("Constant term");
    end if;                            -- if expression statement
  elsif (NAME) then
    if (IDENTIFIER_TAIL) then
      return (TRUE);
    else
      SYNTAX_ERROR("Constant term");
    end if;                            -- if identifier_tail statement
  else
    return (FALSE);
  end if;                      -- if bypass(token_array)
end CONSTANT_TERM;

--------------------------------------------------------------------------

  -- IDENTIFIER_TAIL -->  CONSTRAINT ?  := EXPRESSION ? ;
  --                 -->  renames NAME ? ;
function IDENTIFIER_TAIL return boolean is
begin
put(RESULT_FILE, "IN IDENTIFIER TAIL"); NEW_LINE(RESULT_FILE);
  if (CONSTRAINT) then
    null;
  end if;                      -- if constraint statement
  if (BYPASS(TOKEN_RENAMES)) then
    if (NAME) then
      null;
    else
      SYNTAX_ERROR("Identifier tail");
    end if;                            -- if name statement
  end if;                      -- if bypass(token_renames)
  if (BYPASS(TOKEN_ASSIGNMENT)) then
    if (EXPRESSION) then
      null;
    else
      SYNTAX_ERROR("Identifier tail");
    end if;                            -- if expression statement
  end if;                      -- if bypass(token_assignment)
  if (BYPASS(TOKEN_SEMICOLON)) then
    return (TRUE);
  else
```

```
  return (FALSE);
 end if;                        -- if bypass(token_semicolon)
end IDENTIFIER_TAIL;

---------------------------------------------------------------------------

 -- PARAMETER_SPECIFICATION --> IDENTIFIER_LIST : MODE NAME [:= EXPRESSION ?]
function PARAMETER_SPECIFICATION return boolean is
begin
put(RESULT_FILE, "IN PARAMETER SPECIFICATION"); NEW_LINE(RESULT_FILE);
 HENRY_WRITE_ENABLE := TRUE;  --to capture first parameter
 if (IDENTIFIER_LIST) then
  if (BYPASS(TOKEN_COLON)) then
   if (MODE) then
    if (NAME) then                  -- check for type_mark
     if (BYPASS(TOKEN_ASSIGNMENT)) then
      if (EXPRESSION) then
       null;
      else
       SYNTAX_ERROR("Parameter specification");
      end if;                   -- if expression statement
     end if;                    -- if bypass(token_assignment)
     return (TRUE);
    else
     SYNTAX_ERROR("Parameter specification");
    end if;                     -- if name statement
   else
    SYNTAX_ERROR("Parameter specification");
   end if;                      -- if mode statement
  else
   SYNTAX_ERROR("Parameter specification");
  end if;                       -- if bypass(token_colon)
 else
  return (FALSE);
 end if;                        -- if identifier_list statement
end PARAMETER_SPECIFICATION;

---------------------------------------------------------------------------

 -- IDENTIFIER_LIST --> identifier { , identifier }*
function IDENTIFIER_LIST return boolean is
begin
put(RESULT_FILE, "IN IDENTIFIER LIST"); NEW_LINE(RESULT_FILE);
 if (BYPASS(TOKEN_IDENTIFIER)) then
  if FORMAL_PARAM_DECLARE AND PACKAGE_BODY_DECLARE then
    WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, PARAM_TYPE, NONE, LAST_RECORD);
  elsif (NOT PACKAGE_BODY_DECLARE) then
   WRITE_HENRY_DATA(LOCAL_DECLARE, DUMMY_LEXEME, IDENT_TYPE,
            NONE, LAST_RECORD);
  end if;
   while (BYPASS(TOKEN_COMMA)) loop
    if (IDENT_DECLARE) OR (FORMAL_PARAM_DECLARE AND PACKAGE_BODY_DECLARE)
```

```
    then
      HENRY_WRITE_ENABLE := TRUE;
    end if;
    if FORMAL_PARAM_DECLARE AND PACKAGE_BODY_DECLARE then
      WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, PARAM_TYPE,
                NONE, NEXT_HEN);
    elsif (NOT FORMAL_PARAM_DECLARE) then
     WRITE_HENRY_DATA(LOCAL_DECLARE, DUMMY_LEXEME, IDENT_TYPE,
                NONE, NEXT_HEN);
    end if;
    if not (BYPASS(TOKEN_IDENTIFIER)) then
     SYNTAX_ERROR("Identifier list");
    end if;              -- if not bypass(token_identifer) statement
  end loop;
  return (TRUE);
 else
  return (FALSE);
 end if;                 -- if bypass(token_identifier) statement
end IDENTIFIER_LIST;

---------------------------------------------------------------------------

 -- MODE --> in ?
 --       --> in out
 --       --> out
function MODE return boolean is
begin
put(RESULT_FILE, "IN PARAMETER MODE"); NEW_LINE(RESULT_FILE);
 if (BYPASS(TOKEN_IN)) then
    if PACKAGE_BODY_DECLARE THEN
    WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, PARAM_TYPE, IN_TYPE, LAST_RECORD
    end if;
   if (BYPASS(TOKEN_OUT)) then
    if PACKAGE_BODY_DECLARE then
     WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, PARAM_TYPE,
                IN_OUT_TYPE, LAST_RECORD);
    end if;
   end if;
 elsif (BYPASS(TOKEN_OUT)) then
  if PACKAGE_BODY_DECLARE then
   WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, PARAM_TYPE, OUT_TYPE, LAST_RECOR
  end if;
 end if;
 if (LAST_RECORD.TYPE_DEFINE = PARAM_TYPE)
   AND (LAST_RECORD.PARAM_TYPE = NONE) THEN
   WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, PARAM_TYPE, IN_TYPE, LAST_RECORD
 end if;
 return (TRUE);
end MODE;

---------------------------------------------------------------------------
```

```
  -- DESIGNATOR --> identifier
  --             --> string_literal
function DESIGNATOR return boolean is
begin
  if (BYPASS(TOKEN_IDENTIFIER)) then
    return (TRUE);
  elsif (BYPASS(TOKEN_STRING_LITERAL)) then
    return (TRUE);
  else
    return (FALSE);
  end if;
end DESIGNATOR;

-----------------------------------------------------------------------------

  -- SIMPLE_STATEMENT --> null ;
  --                  --> ASSIGNMENT_OR_PROCEDURE_CALL
  --                  --> exit EXIT_STATEMENT
  --                  --> return RETURN_STATEMENT
  --                  --> goto GOTO_STATEMENT
  --                  --> delay DELAY_STATEMENT
  --                  --> abort ABORT_STATEMENT
  --                  --> raise RAISE_STATEMENT
function SIMPLE_STATEMENT return boolean is
begin
  if (BYPASS(TOKEN_NULL)) then
    if (BYPASS(TOKEN_SEMICOLON)) then
      return (TRUE);
    else
      SYNTAX_ERROR("Simple statement");
    end if;
  elsif (ASSIGNMENT_OR_PROCEDURE_CALL) then -- includes a check for a
    return (TRUE);                          -- code statement and an
                                            -- entry call statement.
  elsif (BYPASS(TOKEN_EXIT)) then
    if (EXIT_STATEMENT) then
      return (TRUE);
    else
      SYNTAX_ERROR("Simple statement");
    end if;
  elsif (BYPASS(TOKEN_RETURN)) then
    if (RETURN_STATEMENT) then
      return (TRUE);
    else
      SYNTAX_ERROR("Simple statement");
    end if;
  elsif (BYPASS(TOKEN_GOTO)) then
    if (GOTO_STATEMENT) then
      return (TRUE);
    else
      SYNTAX_ERROR("Simple statement");
    end if;
```

```
  elsif (BYPASS(TOKEN_DELAY)) then
    if (DELAY_STATEMENT) then
      return (TRUE);
    else
      SYNTAX_ERROR("Simple statement");
    end if;
  elsif (BYPASS(TOKEN_ABORT)) then
    if (ABORT_STATEMENT) then
      return (TRUE);
    else
      SYNTAX_ERROR("Simple statement");
    end if;
  elsif (BYPASS(TOKEN_RAISE)) then
    if (RAISE_STATEMENT) then
      return (TRUE);
    else
      SYNTAX_ERROR("Simple statement");
    end if;
  else
    return (FALSE);
  end if;
end SIMPLE_STATEMENT;

-----------------------------------------------------------------------------

  -- ASSIGNMENT_OR_PROCEDURE_CALL --> NAME := EXPRESSION ;
  --                              --> NAME ;
function ASSIGNMENT_OR_PROCEDURE_CALL return boolean is

ASSIGN_POINTER, FUNCALL_POINTER : POINTER;

begin
put(result_file, "in assign or procedure call"); new_line(result_file);
  HENRY_WRITE_ENABLE := TRUE;
  ASSIGN_POINTER := NEXT_HEN;
  if (NAME) then
    if (BYPASS(TOKEN_ASSIGNMENT)) then
      ASSIGN_STATEMENT := TRUE;
      WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, ASSIGN_TYPE,
                NONE, NEXT_HEN);
      CREATE_NODE(NEXT_HEN, LAST_RECORD);
     if NAME_TAIL_SET then
        WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, PROCALL_OR_DS,
                NONE, ASSIGN_POINTER);
     end if;
     FUNCALL_POINTER := NEXT_HEN;
     HENRY_WRITE_ENABLE := TRUE;
      if (EXPRESSION) then
        if (BYPASS(TOKEN_SEMICOLON)) then
          NAME_TAIL_SET := FALSE;
          ASSIGN_STATEMENT := FALSE;
          WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, END_ASSIGN_TYPE,
```

```
                NONE, NEXT_HEN);
        CREATE_NODE(NEXT_HEN, LAST_RECORD);
        HENRY_WRITE_ENABLE := FALSE;
        return (TRUE);          -- parsed an assignment statement
       else
        SYNTAX_ERROR("Assignment or procedure call");
       end if;                  -- if bypass(token_semicolon)
      else
       SYNTAX_ERROR("Assignment or procedure call");
      end if;                   -- if expression statement
     elsif (BYPASS(TOKEN_SEMICOLON)) then
      WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, PROCALL_OR_DS,
               NONE, ASSIGN_POINTER);
      CREATE_NODE(NEXT_HEN, LAST_RECORD);
      return (TRUE);                 -- parsed a procedure call statement
     else
      SYNTAX_ERROR("Assignment or procedure call");
     end if;                    -- if bypass(token_assignment)
    else
     return (FALSE);
    end if;                     -- if name statement
  end ASSIGNMENT_OR_PROCEDURE_CALL;

  ----------------------------------------------------------------------------

   -- LABEL --> << identifier >>
  function LABEL return boolean is
  begin
   if (BYPASS(TOKEN_LEFT_BRACKET)) then
    if (BYPASS(TOKEN_IDENTIFIER)) then
     if (BYPASS(TOKEN_RIGHT_BRACKET)) then
      return (TRUE);
     else
      SYNTAX_ERROR("Label");
     end if;                         -- if bypass(token_right_bracket)
    else
     SYNTAX_ERROR("Label");
    end if;                          -- if bypasss(token_identifier)
   else
    return (FALSE);
   end if;                      -- if bypass(token_left_bracket)
  end LABEL;

  ----------------------------------------------------------------------------

   -- ENTRY_DECLARATION --> entry identifier (DISCRETE_RANGE) ?
   --                        FORMAL_PART ? ;
  function ENTRY_DECLARATION return boolean is
  begin
   if (BYPASS(TOKEN_ENTRY)) then
    if (BYPASS(TOKEN_IDENTIFIER)) then
     if (BYPASS(TOKEN_LEFT_PAREN)) then
```

```
      if (DISCRETE_RANGE) then
       if (BYPASS(TOKEN RIGHT_PAREN)) then
        null;
       else
        SYNTAX_ERROR("Entry declaration");
       end if;                      -- if bypass(token_right_paren)
      else
       SYNTAX_ERROR("Entry declaration");
      end if;                              -- if discrete_range statement
     end if;                               -- if bypass(token_left_paren)
     if (FORMAL_PART) then
      null;
     end if;                               -- if formal_part statement
     if (BYPASS(TOKEN_SEMICOLON)) then
      return (TRUE);
     else
      SYNTAX_ERROR("Entry declaration");
     end if;                               -- if bypass(token_semicolon)
    else
     SYNTAX_ERROR("Entry declaration");
    end if;                                -- if bypass(token_identifier)
  else
   return (FALSE);
  end if;                       -- if bypass(token_entry)
end ENTRY_DECLARATION;

---------------------------------------------------------------------------

 -- REPRESENTATION_CLAUSE --> for NAME use record RECORD_REPRESENTATION_CLAUSE
 --                       --> for NAME use [at ?] SIMPLE_EXPRESSION;
function REPRESENTATION_CLAUSE return boolean is
begin
 if (BYPASS(TOKEN_FOR)) then
  if (NAME) then
   if (BYPASS(TOKEN_USE)) then
    if (BYPASS(TOKEN_RECORD_STRUCTURE)) then
     if (RECORD_REPRESENTATION_CLAUSE) then
      return (TRUE);
     else
      SYNTAX_ERROR("Representation clause");
     end if;                -- if record_representation_clause
    elsif (BYPASS(TOKEN_AT)) then
     if (SIMPLE_EXPRESSION) then
      if (BYPASS(TOKEN_SEMICOLON)) then
       return (TRUE);
      else
       SYNTAX_ERROR("Representation clause");
      end if;               -- if bypass(token_semicolon)
     else
      SYNTAX_ERROR("Representation clause");
     end if;                -- if simple_expression statement
    elsif (SIMPLE_EXPRESSION) then
```

```
      if (BYPASS(TOKEN_SEMICOLON)) then
        return (TRUE);
      else
        SYNTAX_ERROR("Representation clause");
      end if;                    -- if bypass(token_semicolon)
     else
      SYNTAX_ERROR("Representation clause");
     end if;                           -- if bypass(token_record)
    else
     SYNTAX_ERROR("Representation clause");
    end if;                            -- if bypass(token_use)
   else
    SYNTAX_ERROR("Representation clause");
   end if;                             -- if name statement
  else
   return (FALSE);
  end if;                       -- if bypass(token_for)
end REPRESENTATION_CLAUSE;

---------------------------------------------------------------------------

 -- RECORD_REPRESENTATION_CLAUSE --> [at mod SIMPLE_EXPRESSION ?
 --                                  NAME at SIMPLE_EXPRESSION range RANGES *
 --                                 end record ;
function RECORD_REPRESENTATION_CLAUSE return boolean is
begin
 if (BYPASS(TOKEN_AT)) then
  if (BYPASS(TOKEN_MOD)) then
   if (SIMPLE_EXPRESSION) then
     null;
   else
     SYNTAX_ERROR("Record representation clause");
   end if;                             -- if simple_expression
  else
   SYNTAX_ERROR("Record representation clause");
  end if;                              -- if bypass(token_mod)
 end if;                        -- if bypass(token_at)
 while (NAME) loop
  if (BYPASS(TOKEN_AT)) then
   if (SIMPLE_EXPRESSION) then
    if (BYPASS(TOKEN_RANGE)) then
     if (RANGES) then
       null;
     else
       SYNTAX_ERROR("Record representation clause");
     end if;                   -- if ranges statement
    else
     SYNTAX_ERROR("Record representation clause");
    end if;                            -- if bypass(token_range)
   else
    SYNTAX_ERROR("Record representation clause");
   end if;                             -- if simple_expression
```

```
    else
      SYNTAX_ERROR("Record representation clause");
    end if;                               -- if bypass(token_at)
  end loop;
  if (BYPASS(TOKEN_END)) then
    if (BYPASS(TOKEN_RECORD_STRUCTURE)) then
      if (BYPASS(TOKEN_SEMICOLON)) then
        return (TRUE);
      else
        SYNTAX_ERROR("Record representation clause");
      end if;                             -- if bypass(token_semicolon)
    else
      SYNTAX_ERROR("Record representation clause");
    end if;                               -- if bypass(token_record_structure)
  else
    return (FALSE);
  end if;                                 -- if bypass(token_end)
end RECORD_REPRESENTATION_CLAUSE;

end PARSER_2;


--**************************************************************--
--
-- TITLE:        AN ADA SOFTWARE METRIC
--
-- MODULE NAME:   PACKAGE PARSER_3
-- DATE CREATED:  22 JUL 86
-- LAST MODIFIED: 30 MAY 87
--
-- AUTHORS:       LCDR JEFFREY L. NIEDER
--                LT KARL S. FAIRBANKS, JR.
--                LCDR PAUL M. HERZIG
-- DESCRIPTION:  This package contains thirty-five functions
--     that make up the baseline productions for our top-down,
--     recursive descent parser.  Each function is preceded
--     by the grammar productions they are implementing.
--
--**************************************************************--

with PARSER_4, HENRY_GLOBAL, HENRY, BYPASS_FUNCTION, HALSTEAD_METRIC,
    GLOBAL_PARSER, GLOBAL, TEXT_IO;
use PARSER_4, HENRY_GLOBAL, HENRY, BYPASS_FUNCTION, HALSTEAD_METRIC,
    GLOBAL_PARSER, GLOBAL, TEXT_IO;

package PARSER_3 is
  function SUBTYPE_INDICATION return boolean;
  function ARRAY_TYPE_DEFINITION return boolean;
  function CHOICE return boolean;
  function ITERATION_SCHEME return boolean;
  function LOOP_PARAMETER_SPECIFICATION return boolean;
```

```
  function EXPRESSION return boolean;
  function RELATION return boolean;
  function RELATION_TAIL return boolean;
  function SIMPLE_EXPRESSION return boolean;
  function SIMPLE_EXPRESSION_TAIL return boolean;
  function TERM return boolean;
  function FACTOR return boolean;
  function PRIMARY return boolean;
  function CONSTRAINT return boolean;
  function FLOATING_OR_FIXED_POINT_CONSTRAINT return boolean;
  function INDEX_CONSTRAINT return boolean;
  function RANGES return boolean;
  function AGGREGATE return boolean;
  function COMPONENT_ASSOCIATION return boolean;
  function ALLOCATOR return boolean;
  function NAME return boolean;
  function NAME_TAIL return boolean;
  function LEFT_PAREN_NAME_TAIL return boolean;
  function ATTRIBUTE_DESIGNATOR return boolean;
  function INTEGER_TYPE_DEFINITION return boolean;
  function DISCRETE_RANGE return boolean;
  function EXIT_STATEMENT return boolean;
  function RETURN_STATEMENT return boolean;
  function GOTO_STATEMENT return boolean;
  function DELAY_STATEMENT return boolean;
  function ABORT_STATEMENT return boolean;
  function RAISE_STATEMENT return boolean;
end PARSER_3;

--------------------------------------------------------------------------
--------------------------------------------------------------------------

package body PARSER_3 is

  -- SUBTYPE_INDICATION --> NAME CONSTRAINT ?
function SUBTYPE_INDICATION return boolean is
begin
  if (NAME) then                          -- check for type_mark
    if (CONSTRAINT) then
      null;
    end if;
    return (TRUE);
  else
    return (FALSE);
  end if;
end SUBTYPE_INDICATION;

--------------------------------------------------------------------------

  -- ARRAY_TYPE_DEFINITION --> (INDEX_CONSTRAINT of SUBTYPE_INDICATION
  -- this function parses both constrained and unconstrained arrays
function ARRAY_TYPE_DEFINITION return boolean is
```

```
begin
  if (BYPASS(TOKEN LEFT PAREN)) then
    if (INDEX CONSTRAINT) then
      if (BYPASS(TOKEN OF)) then
        if (SUBTYPE INDICATION) then
          return (TRUE);
        else
          SYNTAX_ERROR("Array definition");
        end if;                    -- if subtype_indication
      else
        SYNTAX_ERROR("Array definition");
      end if;                      -- if bypass(token_of)
    else
      SYNTAX ERROR("Array definition");
    end if;                        -- if index_constraint statement
  else
    return (FALSE);
  end if;                          -- if bypass(token_left_paren)
end ARRAY TYPE_DEFINITION;

------------------------------------------------------------------------

  -- CHOICE --> EXPRESSION ..SIMPLE_EXPRESSION ?
  --        --> EXPRESSION CONSTRAINT ?
  --        --> others
function CHOICE return boolean is
begin
  if (EXPRESSION) then
    if (BYPASS(TOKEN_RANGE_DOTS)) then -- check for discrete range
      if (SIMPLE_EXPRESSION) then
        null;
      else
        SYNTAX_ERROR("Choice");
      end if;                      -- if simple_expression statement
    elsif (CONSTRAINT) then
      null;
    end if;                        -- if bypass token_range_dots
    return (TRUE);
  elsif (BYPASS(TOKEN_OTHERS)) then
    return (TRUE);
  else
    return (FALSE);
  end if;
end CHOICE;

------------------------------------------------------------------------

  -- ITERATION SCHEME --> while EXPRESSION
  --                  --> for LOOP PARAMETER SPECIFICATION
function ITERATION SCHEME return boolean is
begin
  if (BYPASS(TOKEN WHILE)) then
```

```
    NESTING_METRIC(WHILE_CONSTRUCT);
    if (EXPRESSION) then
      return (TRUE);
    else
      SYNTAX_ERROR("Iteration scheme");
    end if;
  elsif (BYPASS(TOKEN_FOR)) then
    NESTING_METRIC(FOR_CONSTRUCT);
    if (LOOP_PARAMETER_SPECIFICATION) then
      return (TRUE);
    else
      SYNTAX_ERROR("Iteration scheme");
    end if;
  else
    return (FALSE);
  end if;
end ITERATION_SCHEME;

------------------------------------------------------------------------

 -- LOOP_PARAMETER_SPECIFICATION --> identifier in reverse ? DISCRETE_RANGE
function LOOP_PARAMETER_SPECIFICATION return boolean is
begin
 if (BYPASS(TOKEN_IDENTIFIER)) then
   if (BYPASS(TOKEN_IN)) then
     if (BYPASS(TOKEN_REVERSE)) then
       null;
     end if;                    -- if bypass(token_reverse)
     if (DISCRETE_RANGE) then
       return (TRUE);
     else
       SYNTAX_ERROR("Loop parameter specification");
     end if;                    -- if discrete_range statement
   else
     SYNTAX_ERROR("Loop parameter specification");
   end if;                      -- if bypass(token_in)
 else
   return (FALSE);
 end if;                        -- if bypass(token_identifier)
end LOOP_PARAMETER_SPECIFICATION;

------------------------------------------------------------------------

 -- EXPRESSION --> RELATION RELATION_TAIL ?
function EXPRESSION return boolean is
begin
 if (RELATION) then
   if (RELATION_TAIL) then
     null;
   end if;                      -- if relation_tail statement
   return (TRUE);
 else
```

```
    return (FALSE);
  end if;                      -- if relation statement
end EXPRESSION;

--------------------------------------------------------------------------

  -- RELATION --> SIMPLE_EXPRESSION SIMPLE_EXPRESSION_TAIL ?
function RELATION return boolean is
begin
  if (SIMPLE_EXPRESSION) then
    if (SIMPLE_EXPRESSION_TAIL) then
      null;
    end if;                    -- if simple_expression_tail statement
    return (TRUE);
  else
    return (FALSE);
  end if;                      -- if simple_expression statement
end RELATION;

--------------------------------------------------------------------------

  -- RELATION_TAIL --> [and [then ?] RELATION]*
  --               --> [or [else ?] RELATION]*
  --               --> [xor RELATION]*
function RELATION_TAIL return boolean is
begin
  while (BYPASS(TOKEN_AND)) loop
    if (BYPASS(TOKEN_THEN)) then
      null;
    end if;                              -- if bypass(token_then)
    if not (RELATION) then
      SYNTAX_ERROR("Relation tail");
    end if;                              -- if not relation statement
  end loop;
  while (BYPASS(TOKEN_OR)) loop
    if (BYPASS(TOKEN_ELSE)) then
      null;
    end if;                              -- if bypass(token_else)
    if not (RELATION) then
      SYNTAX_ERROR("Relation tail");
    end if;                              -- if not relation statement
  end loop;
  while (BYPASS(TOKEN_XOR)) loop
    if not (RELATION) then
      SYNTAX_ERROR("Relation tail");
    end if;                              -- if not relation statement
  end loop;
  return (TRUE);
end RELATION_TAIL;

--------------------------------------------------------------------------
```

```
 -- SIMPLE_EXPRESSION --> - ? TERM BINARY ADDING_OPERATOR TERM *
 --                 --> - ? TERM BINARY ADDING OPERATOR TERM *
function SIMPLE_EXPRESSION return boolean is
begin
 if (BYPASS(TOKEN_PLUS) or BYPASS(TOKEN_MINUS)) then
  if (TERM) then
   while (BINARY_ADDING_OPERATOR) loop
    if not (TERM) then
     SYNTAX_ERROR("Simple expression");
    end if;            -- if not term statement
   end loop;
   return (TRUE);
  else
   SYNTAX_ERROR("Simple expression");
  end if;              -- if term statement
 elsif (TERM) then
  while (BINARY_ADDING_OPERATOR) loop
   if not (TERM) then
    SYNTAX_ERROR("Simple expression");
   end if;             -- if not term statement
  end loop;
  return (TRUE);
 else
  return (FALSE);
 end if;               -- if bypass(token_plus) et al statement
end SIMPLE_EXPRESSION;

------------------------------------------------------------------------

 -- SIMPLE_EXPRESSION_TAIL --> RELATIONAL_OPERATOR SIMPLE_EXPRESSION
 --                      --> [not ?] in RANGES
 --                      --> [not ?] in NAME
function SIMPLE_EXPRESSION_TAIL return boolean is
begin
 if (RELATIONAL_OPERATOR) then
  if (SIMPLE_EXPRESSION) then
   return (TRUE);
  else
   SYNTAX_ERROR("Simple expression tail");
  end if;                -- if simple_expression statement
 elsif (BYPASS(TOKEN_NOT)) then
  if (BYPASS(TOKEN_IN)) then
   if (RANGES) then
    return (TRUE);
   elsif (NAME) then             -- check for type_mark
    return (TRUE);
   else
    SYNTAX_ERROR("Simple expression tail");
   end if;               -- if ranges statement
  else
   SYNTAX_ERROR("Simple expression tail");
  end if;                -- if bypass(token_in) statement
```

```
 elsif (BYPASS(TOKEN_IN)) then
   if (RANGES) then
     return (TRUE);
   elsif (NAME) then                  -- check for type_mark
     return (TRUE);
   else
     SYNTAX_ERROR("Simple expression tail");
   end if;                            -- if ranges statement
 else
   return (FALSE);
 end if;                              -- if relational_operator statement
end SIMPLE_EXPRESSION_TAIL;

---------------------------------------------------------------------------

 -- TERM --> FACTOR {MULTIPLYING_OPERATOR FACTOR}*
function TERM return boolean is
begin
 if (FACTOR) then
   while (MULTIPLYING_OPERATOR) loop
     if not (FACTOR) then
       SYNTAX_ERROR("Term");
     end if;                    -- if not factor statement
   end loop;
   return (TRUE);
 else
   return (FALSE);
 end if;                        -- if factor statement
end TERM;

---------------------------------------------------------------------------

 -- FACTOR --> PRIMARY [** PRIMARY]?
 --         --> abs PRIMARY
 --         --> not PRIMARY
function FACTOR return boolean is
begin
 if (PRIMARY) then
   if (BYPASS(TOKEN_EXPONENT)) then
     if (PRIMARY) then
       null;
     else
       SYNTAX_ERROR("Factor");
     end if;                    -- if primary statement
   end if;                      -- if bypass(token_exponent) statement
   return (TRUE);
 elsif (BYPASS(TOKEN_ABSOLUTE)) then
   if (PRIMARY) then
     return (TRUE);
   else
     SYNTAX_ERROR("Factor");
   end if;                      -- if primary(abs) statement
```

```
  elsif (BYPASS(TOKEN_NOT)) then
   if (PRIMARY) then
    return (TRUE);
   else
    SYNTAX_ERROR("Factor");
   end if;                    -- if primary(not) statement
  else
   return (FALSE);
  end if;                     -- if primary statement
end FACTOR;

-----------------------------------------------------------------------------

 -- PRIMARY --> numeric_literal
 --          --> null
 --          --> string_literal
 --          --> new ALLOCATOR
 --          --> NAME
 --        --> AGGREGATE
function PRIMARY return boolean is
begin
 HENRY_WRITE_ENABLE := TRUE;
 if (BYPASS(TOKEN_NUMERIC_LITERAL)) then
  WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, IDENT_TYPE, NONE, LAST_RECORD);
  return (TRUE);
 elsif (BYPASS(TOKEN_NULL)) then
  return (TRUE);
 elsif (BYPASS(TOKEN_STRING_LITERAL)) then
  WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, IDENT_TYPE, NONE, LAST_RECORD);
  return (TRUE);
 elsif (BYPASS(TOKEN_NEW)) then
  if (ALLOCATOR) then
   return (TRUE);
  else
   SYNTAX_ERROR("Primary");
  end if;                               -- if allocator statement
 elsif (NAME) then
  return (TRUE);
 elsif (AGGREGATE) then
  return (TRUE);
 else
  return (FALSE);
 end if;                     -- if bypass(token_left_paren)
end PRIMARY;

-----------------------------------------------------------------------------

 -- CONSTRAINT --> range RANGES
 --             --> range <>
 --             --> digits FLOATING_OR_FIXED_POINT_CONSTRAINT
 --             --> delta FLOATING_OR_FIXED_POINT_CONSTRAINT
 --             --> (INDEX_CONSTRAINT
```

```
function CONSTRAINT return boolean is
begin
  if (BYPASS(TOKEN_RANGE)) then
    if (RANGES) then
      return (TRUE);
    elsif (BYPASS(TOKEN_BRACKETS)) then          -- check for <> when parsing
      return (TRUE);                             --  an unconstrained array
    else
      SYNTAX_ERROR("Constraint");
    end if;                                      -- if ranges statement
  elsif (BYPASS(TOKEN_DIGITS)) or else (BYPASS(TOKEN_DELTA)) then
    if (FLOATING_OR_FIXED_POINT_CONSTRAINT) then
      return (TRUE);
    else
      SYNTAX_ERROR("Constraint");
    end if;
  elsif (BYPASS(TOKEN_LEFT_PAREN)) then
    if (INDEX_CONSTRAINT) then
      return (TRUE);
    else
      SYNTAX_ERROR("Constraint");
    end if;
  else
    return (FALSE);
  end if;
end CONSTRAINT;

---------------------------------------------------------------------------

  -- FLOATING_OR_FIXED_POINT_CONSTRAINT --> SIMPLE_EXPRESSION range RANGES ?
function FLOATING_OR_FIXED_POINT_CONSTRAINT return boolean is
begin
  if (SIMPLE_EXPRESSION) then
    if (BYPASS(TOKEN_RANGE)) then
      if (RANGES) then
        null;
      else
        SYNTAX_ERROR("Floating or fixed point constraint");
      end if;                              -- if ranges statement
    end if;                                -- if bypass(token_range)
    return (TRUE);
  else
    return (FALSE);
  end if;                          -- if simple_expression statement
end FLOATING_OR_FIXED_POINT_CONSTRAINT;

---------------------------------------------------------------------------

  -- INDEX_CONSTRAINT --> ( DISCRETE_RANGE , DISCRETE_RANGE * )
function INDEX_CONSTRAINT return boolean is
begin
  if (DISCRETE_RANGE) then
```

```
    while (BYPASS(TOKEN_COMMA)) loop
      if not (DISCRETE_RANGE) then
        SYNTAX_ERROR("Index constraint");
      end if;                              -- if not discrete_range
    end loop;
    if (BYPASS(TOKEN_RIGHT_PAREN)) then
      return (TRUE);
    else
      SYNTAX_ERROR("Index constraint");
    end if;                                -- if bypass(token_right_paren)
  else
    return (FALSE);
  end if;                        -- if discrete_range statement
end INDEX_CONSTRAINT;

------------------------------------------------------------------------------

  -- RANGES --> SIMPLE_EXPRESSION ..SIMPLE_EXPRESSION ?
function RANGES return boolean is
begin
  if (SIMPLE_EXPRESSION) then
    if (BYPASS(TOKEN_RANGE_DOTS)) then
      if (SIMPLE_EXPRESSION) then
        null;
      else
        SYNTAX_ERROR("Ranges");
      end if;                    -- if simple_expression statement
    end if;                            -- if bypass(token_range_dots)
    return (TRUE);
  else
    return (FALSE);
  end if;                        -- if simple_expression statement
end RANGES;

------------------------------------------------------------------------------

  -- AGGREGATE --> (COMPONENT_ASSOCIATION ,COMPONENT_ASSOCIATION * )
function AGGREGATE return boolean is
begin
  if (BYPASS(TOKEN_LEFT_PAREN)) then
    if (COMPONENT_ASSOCIATION) then
      while (BYPASS(TOKEN_COMMA)) loop
        if not (COMPONENT_ASSOCIATION) then
          SYNTAX_ERROR("Aggregate");
        end if;                        -- if not component_association
      end loop;
      if (BYPASS(TOKEN_RIGHT_PAREN)) then
        return (TRUE);
      else
        SYNTAX_ERROR("Aggregate");
      end if;                          -- if bypass(token_right_paren)
    else
```

```
    SYNTAX ERROR("Aggregate");
   end if;                                -- if component  association statement
  else
   return (FALSE);
  end if;                     -- if bypass(token  left  paren)
end AGGREGATE;

--------------------------------------------------------------------------

 -- COMPONENT  ASSOCIATION --> CHOICE   CHOICE *    ? EXPRESSION
function COMPONENT  ASSOCIATION return boolean is
begin
 if (CHOICE) then
   while (BYPASS(TOKEN  BAR)) loop
    if not (CHOICE) then
     SYNTAX  ERROR("Component asociation");
    end if;
   end loop;
   if (BYPASS(TOKEN  ARROW)) then
    if (EXPRESSION) then
     null;
    else
     SYNTAX  ERROR("Component asociation");
    end if;                               -- if expression statement
   end if;                                -- if bypass(token  arrow)
   return (TRUE);
 else
   return (FALSE);
 end if;                      -- if choice statement
end COMPONENT  ASSOCIATION;

--------------------------------------------------------------------------

 -- ALLOCATOR --> SUBTYPE  INDICATION 'AGGREGATE ?
function ALLOCATOR return boolean is
begin
 if (SUBTYPE  INDICATION) then
   if (BYPASS(TOKEN  APOSTROPHE)) then
    if (AGGREGATE) then
     null;
    else
     SYNTAX  ERROR("Allocator");
    end if;                               -- if aggregate statement
   end if;                                -- if bypass(token  apostrophe)
   return (TRUE);
 else
   return (FALSE);
 end if;                      -- if subtype  indication statement
end ALLOCATOR;

--------------------------------------------------------------------------
```

```
-- NAME --> identifier [NAME_TAIL ?]
--       --> character_literal [NAME_TAIL ?]
--       --> string_literal [NAME_TAIL ?]
function NAME return boolean is


begin
put(result_file, "in name"); new_line(result_file);
 if (BYPASS(TOKEN_IDENTIFIER)) then
   NAME_POINTER := LAST_RECORD;
  if (NAME_TAIL) then
    null;
  end if;
  return (TRUE);
 HENRY_WRITE_ENABLE := TRUE;
 elsif (BYPASS(TOKEN_CHARACTER_LITERAL)) then
  if (NAME_TAIL) then
   null;
  end if;
  return (TRUE);
 elsif (BYPASS(TOKEN_STRING_LITERAL)) then
  if (NAME_TAIL) then
   null;
  end if;
  return (TRUE);
 else
  return (FALSE);
 end if;
end NAME;

--------------------------------------------------------------------------

 -- NAME_TAIL --> (LEFT_PAREN_NAME_TAIL
 --             --> .SELECTOR [NAME_TAIL *]
 --             --> 'AGGREGATE [NAME_TAIL *]
 --             --> 'ATTRIBUTE_DESIGNATOR [NAME_TAIL *]
function NAME_TAIL return boolean is
begin
put(result_file, "in name tail"); new_line(result_file);
 if (BYPASS(TOKEN_LEFT_PAREN)) then
   NAME_TAIL_SET := TRUE;
   HENRY_WRITE_ENABLE := TRUE;
   if ASSIGN_STATEMENT then
    WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, FUNCALL_OR_DS,
              NONE, NAME_POINTER);
   else WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, PROCALL_OR_DS,
                NONE, NAME_POINTER);
   end if;
  if (LEFT_PAREN_NAME_TAIL) then
   return (TRUE);
  else
   return (FALSE);
```

```
  end if;                    -- if left_paren_name_tail
 elsif (BYPASS(TOKEN_PERIOD)) then
  if (SELECTOR) then
   while (NAME_TAIL) loop
    null;
   end loop;
   return (TRUE);
  else
   SYNTAX_ERROR("Name tail");
  end if;                    -- if selector statement
 elsif (BYPASS(TOKEN_APOSTROPHE)) then
  if (AGGREGATE) then
   while (NAME_TAIL) loop
    null;
   end loop;
   return (TRUE);
  elsif (ATTRIBUTE_DESIGNATOR) then
   while (NAME_TAIL) loop
    null;
   end loop;
   return (TRUE);
  else
   SYNTAX_ERROR("Name tail");
  end if;                    -- if aggregate statement
 else
  return (FALSE);
 end if;                     -- if bypass(token_left_paren)
end NAME_TAIL;

----------------------------------------------------------------------------

 -- LEFT_PAREN_NAME_TAIL --> [FORMAL_PARAMETER ? EXPRESSION [..EXPRESSION ?
 --                  , FORMAL_PARAMETER ? EXPRESSION [..EXPRESSION ? ]*
 --              ) NAME_TAIL *
function LEFT_PAREN_NAME_TAIL return boolean is
begin
put(result_file, "in left paren name tail"); new_line(result_file);
 if (FORMAL_PARAMETER) then          -- check for optional formal parameter
  null;                     --  before the actual parameter
 end if;                    -- if formal_parameter statement
 HENRY_WRITE_ENABLE := TRUE;
 if (EXPRESSION) then
  if NAME_TAIL_SET then
   WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, PARAM_TYPE, ACTUAL_PARAM,
           LAST_RECORD);
  end if;
  if (BYPASS(TOKEN_RANGE_DOTS)) then
   if not (EXPRESSION) then
    SYNTAX_ERROR("Left paren name tail");
   end if;                        -- if not expression statement
  end if;                         -- if bypass(token_range_dots)
  while (BYPASS(TOKEN_COMMA)) loop
```

```
    if (FORMAL_PARAMETER) then
      null;
    end if;                          -- if formal_parameter statement
   HENRY_WRITE_ENABLE := TRUE;
    if not (EXPRESSION) then
      SYNTAX_ERROR("Left paren name tail");
    end if;                          -- if not expression statement
    if (BYPASS(TOKEN_RANGE_DOTS)) then
      if not (EXPRESSION) then
        SYNTAX_ERROR("Left paren name tail");
      end if;                        -- if not expression statement
    end if;                          -- if bypass(token_range_dots)
    if NAME_TAIL_SET then
    WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, PARAM_TYPE, ACTUAL_PARAM,
              LAST_RECORD);
    end if;
   end loop;
   if (BYPASS(TOKEN_RIGHT_PAREN)) then
    WRITE_HENRY_DATA(BLANK, DUMMY_LEXEME, END_ACTUAL_PARAM,
              ACTUAL_PARAM, NEXT_HEN);
    CREATE_NODE(NEXT_HEN, LAST_RECORD);
    NAME_TAIL_SET := FALSE;
    while (NAME_TAIL) loop
      null;
    end loop;
    return (TRUE);
   else
    return (FALSE);
   end if;               -- if bypass(token_right_paren)
  elsif (DISCRETE_RANGE) then
   if (BYPASS(TOKEN_RIGHT_PAREN)) then
     while (NAME_TAIL) LOOP
      NULL;
     END LOOP;
     RETURN (TRUE);
    else
     SYNTAX_ERROR("Left paren name tail");
    end if;
  else
    return (FALSE);
   end if;               -- if bypass(token_right_paren)
end LEFT_PAREN_NAME_TAIL;

--------------------------------------------------------------------------

 -- ATTRIBUTE_DESIGNATOR --> identifier (EXPRESSION) ?
 --                      --> range (EXPRESSION) ?
 --                      --> digits (EXPRESSION) ?
 --                      --> delta (EXPRESSION) ?
function ATTRIBUTE_DESIGNATOR return boolean is
begin
 if (BYPASS(TOKEN_IDENTIFIER)) or else (BYPASS(TOKEN_RANGE)) then
```

```
   if (BYPASS(TOKEN_LEFT_PAREN)) then
    if (EXPRESSION) then
     if (BYPASS(TOKEN_RIGHT_PAREN)) then
      null;
     else
      SYNTAX_ERROR("Attribute designator");
     end if;                -- if bypass(token_right_paren) statement
    else
     SYNTAX_ERROR("Attribute designator");
    end if;                 -- if expression statement
   end if;                  -- if bypass(token_left_paren) statement
   return (TRUE);
  elsif (BYPASS(TOKEN_DIGITS)) or else (BYPASS(TOKEN_DELTA)) then
   if (BYPASS(TOKEN_LEFT_PAREN)) then
    if (EXPRESSION) then
     if (BYPASS(TOKEN_RIGHT_PAREN)) then
      null;
     else
      SYNTAX_ERROR("Attribute designator");
     end if;                -- if bypass(token_right_paren) statement
    else
     SYNTAX_ERROR("Attribute designator");
    end if;                 -- if expression statement
   end if;                  -- if bypass(token_left_paren) statement
   return (TRUE);
  else
   return (FALSE);
  end if;                   -- if bypass(token_identifier) statement
end ATTRIBUTE_DESIGNATOR;

-----------------------------------------------------------------------------

 -- INTEGER_TYPE_DEFINITION --> range RANGES
function INTEGER_TYPE_DEFINITION return boolean is
begin
 if (BYPASS(TOKEN_RANGE)) then
  if (RANGES) then
   return (TRUE);
  else
   SYNTAX_ERROR("Integer type definition");
  end if;
 else
  return (FALSE);
 end if;
end INTEGER_TYPE_DEFINITION;

-----------------------------------------------------------------------------

 -- DISCRETE_RANGE --> RANGES CONSTRAINT ?
function DISCRETE_RANGE return boolean is
begin
 if (RANGES) then
```

```
    if (CONSTRAINT) then
      null;
    end if;                              -- if constraint statement
    return (TRUE);
  else
    return (FALSE);
  end if;                        -- if ranges statement
end DISCRETE_RANGE;

-----------------------------------------------------------------------

  -- EXIT_STATEMENT --> [NAME] [when EXPRESSION] ;
function EXIT_STATEMENT return boolean is
begin
  if (NAME) then
    null;
  end if;                        -- if name statement
  if (BYPASS(TOKEN_WHEN)) then
    if (EXPRESSION) then
      null;
    else
      SYNTAX_ERROR("Exit statement");
    end if;                              -- if expression statement
  end if;                      -- if bypass(token_when)
  if (BYPASS(TOKEN_SEMICOLON)) then
    return (TRUE);
  else
    return (FALSE);
  end if;                      -- if bypass(token_semicolon)
end EXIT_STATEMENT;

-----------------------------------------------------------------------

  -- RETURN_STATEMENT --> [EXPRESSION] ;
function RETURN_STATEMENT return boolean is
begin
  if (EXPRESSION) then
    null;
  end if;
  if (BYPASS(TOKEN_SEMICOLON)) then
    return (TRUE);
  else
    return (FALSE);
  end if;
end RETURN_STATEMENT;

-----------------------------------------------------------------------

  -- GOTO_STATEMENT --> NAME ;
function GOTO_STATEMENT return boolean is
begin
  if (NAME) then
```

```
    if (BYPASS(TOKEN_SEMICOLON)) then
      return (TRUE);
    else
      SYNTAX_ERROR("Goto statement");
    end if;                        -- if bypass(token_semicolon)
  else
    return (FALSE);
  end if;                          -- if name statement
end GOTO_STATEMENT;

------------------------------------------------------------------------

  -- DELAY_STATEMENT --> SIMPLE_EXPRESSION ;
function DELAY_STATEMENT return boolean is
begin
  if (SIMPLE_EXPRESSION) then
    if (BYPASS(TOKEN_SEMICOLON)) then
      return (TRUE);
    else
      SYNTAX_ERROR("Delay statement");
    end if;                        -- if bypass(token_semicolon)
  else
    return (FALSE);
  end if;                          -- if simple_expression statement
end DELAY_STATEMENT;

------------------------------------------------------------------------

  -- ABORT_STATEMENT --> NAME , NAME * ;
function ABORT_STATEMENT return boolean is
begin
  if (NAME) then
    while (BYPASS(TOKEN_COMMA)) loop
      if not (NAME) then
        SYNTAX_ERROR("Abort statement");
      end if;                      -- if not name statement
    end loop;
    if (BYPASS(TOKEN_SEMICOLON)) then
      return (TRUE);
    else
      SYNTAX_ERROR("Abort statement");
    end if;                        -- if bypass(token_semicolon)
  else
    return (FALSE);
  end if;                          -- if name statement
end ABORT_STATEMENT;

------------------------------------------------------------------------

  -- RAISE_STATEMENT --> NAME ? ;
function RAISE_STATEMENT return boolean is
begin
```

```
 if (NAME) then
   null;
 end if;
 if (BYPASS(TOKEN_SEMICOLON)) then
   return (TRUE);
 else
   return (FALSE);
 end if;
end RAISE_STATEMENT;

end PARSER_3;




--*************************************************************--
--
-- TITLE:          AN ADA SOFTWARE METRIC
--
-- MODULE NAME:   PACKAGE PARSER_4
-- DATE CREATED:  23 JUL 86
-- LAST MODIFIED: 30 MAY 87
--
-- AUTHORS:        LCDR JEFFREY L. NIEDER
--                 LT KARL S. FAIRBANKS, JR.
--                 LCDR PAUL M. HERZIG
-- DESCRIPTION: This package contains seven functions that
--      are the lowest level productions for our top-down,
--      recursive descent parser. Each function is preceded
--      by the grammar productions they are implementing.
--
--*************************************************************--

with BYPASS_FUNCTION, BYPASS_SUPPORT_FUNCTIONS, GLOBAL_PARSER, GLOBAL, TEXT_I
use BYPASS_FUNCTION, BYPASS_SUPPORT_FUNCTIONS, GLOBAL_PARSER, GLOBAL, TEXT_IC

package PARSER_4 is
 function MULTIPLYING_OPERATOR return boolean;
 function BINARY_ADDING_OPERATOR return boolean;
 function RELATIONAL_OPERATOR return boolean;
 function ENUMERATION_TYPE_DEFINITION return boolean;
 function ENUMERATION_LITERAL return boolean;
 function FORMAL_PARAMETER return boolean;
 function SELECTOR return boolean;
end PARSER_4;

-----------------------------------------------------------------------
-----------------------------------------------------------------------


package body PARSER_4 is
```

```
 -- MULTIPLYING_OPERATOR --> *
 --                        -->
 --                        --> mod
 --                        --> rem
function MULTIPLYING_OPERATOR return boolean is
begin
put(RESULT_FILE, "In multiplying_operator "); new_line(RESULT_FILE);
 if (BYPASS(TOKEN_ASTERISK)) then
  return (TRUE);
 elsif (BYPASS(TOKEN_SLASH)) then
  return (TRUE);
 elsif (BYPASS(TOKEN_MOD)) then
  return (TRUE);
 elsif (BYPASS(TOKEN_REM)) then
  return (TRUE);
 else
  return (FALSE);
 end if;
end MULTIPLYING_OPERATOR;

------------------------------------------------------------------------

 -- BINARY_ADDING_OPERATOR --> +
 --                        --> -
 --                        --> &
function BINARY_ADDING_OPERATOR return boolean is
begin
put(RESULT_FILE, "In binary_adding_operator "); new_line(RESULT_FILE);
 if (BYPASS(TOKEN_PLUS)) then
  return (TRUE);
 elsif (BYPASS(TOKEN_MINUS)) then
  return (TRUE);
 elsif (BYPASS(TOKEN_AMPERSAND)) then
  return (TRUE);
 else
  return (FALSE);
 end if;
end BINARY_ADDING_OPERATOR;

------------------------------------------------------------------------

 -- RELATIONAL_OPERATOR --
 --                     --
 --                     --
 --                     --
 --                     --
 --                     --
function RELATIONAL_OPERATOR return boolean is
begin
put(RESULT_FILE, "In relational_operator "); new_line(RESULT_FILE);
 if (BYPASS(TOKEN_EQUALS)) then
  return (TRUE);
```

```
 elsif (BYPASS(TOKEN_NOT_EQUALS)) then
  return (TRUE);
 elsif (BYPASS(TOKEN_LESS_THAN)) then
  return (TRUE);
 elsif (BYPASS(TOKEN_LESS_THAN_EQUALS)) then
  return (TRUE);
 elsif (BYPASS(TOKEN_GREATER_THAN)) then
  return (TRUE);
 elsif (BYPASS(TOKEN_GREATER_THAN_EQUALS)) then
  return (TRUE);
 else
  return (FALSE);
 end if;
end RELATIONAL_OPERATOR;

------------------------------------------------------------------------

 -- ENUMERATION_TYPE_DEFINITION --> (ENUMERATION_LITERAL
 --                                  , ENUMERATION_LITERAL *)
function ENUMERATION_TYPE_DEFINITION return boolean is
begin
put(RESULT_FILE, "In enumeration_type_definition "); new_line(RESULT_FILE);
 if (BYPASS(TOKEN_LEFT_PAREN)) then
   HENRY_WRITE_ENABLE := TRUE;
  if (ENUMERATION_LITERAL) then
   while (BYPASS(TOKEN_COMMA)) loop
    HENRY_WRITE_ENABLE := TRUE;
    if not (ENUMERATION_LITERAL) then
     SYNTAX_ERROR("Enumeration type definition");
    end if;                  -- if not enumeration_literal
   end loop;
   if (BYPASS(TOKEN_RIGHT_PAREN)) then
    return (TRUE);
   else
    SYNTAX_ERROR("Enumeration type definition");
   end if;                   -- if bypass(token_right_paren)
  else
   SYNTAX_ERROR("Enumeration type definition");
  end if;                    -- if enumeration_literal statement
 else
  return (FALSE);
 end if;                     -- if bypass(token_left_paren)
end ENUMERATION_TYPE_DEFINITION;

------------------------------------------------------------------------

 -- ENUMERATION_LITERAL --> identifier
 --                     --> character_literal
function ENUMERATION_LITERAL return boolean is
begin
put(RESULT_FILE, "In enumeration_literal "); new_line(RESULT_FILE);
 if (BYPASS(TOKEN_IDENTIFIER)) then
```

```
    return (TRUE);
  elsif (BYPASS(TOKEN_CHARACTER_LITERAL)) then
    return (TRUE);
  else
    return (FALSE);
  end if;
end ENUMERATION_LITERAL;

-----------------------------------------------------------------------------

  -- FORMAL_PARAMETER --> identifier =>
function FORMAL_PARAMETER return boolean is
begin
put(RESULT_FILE, "In formal_parameter "); new_line(RESULT_FILE);
  LOOK_AHEAD_TOKEN := TOKEN_RECORD_BUFFER(TOKEN_ARRAY_INDEX + 1);
  if (ADJUST_LEXEME(LOOK_AHEAD_TOKEN.LEXEME,
          LOOK_AHEAD_TOKEN.LEXEME_SIZE - 1) = "=>") then
    if (BYPASS(TOKEN_IDENTIFIER)) then
      if (BYPASS(TOKEN_ARROW)) then
        return (TRUE);
      else
        SYNTAX_ERROR("Formal parameter");
      end if;                              -- if bypass(token_arrow)
    else
      SYNTAX_ERROR("Formal parameter");
    end if;                                -- if bypass(token_identifier)
  else
    return (FALSE);
  end if;
end FORMAL_PARAMETER;

-----------------------------------------------------------------------------

  -- SELECTOR --> identifier
  --          --> character_literal
  --          --> string_literal
  --          --> all
function SELECTOR return boolean is
begin
put(RESULT_FILE, "In selector "); new_line(RESULT_FILE);
  if (BYPASS(TOKEN_IDENTIFIER)) then
    return (TRUE);
  elsif (BYPASS(TOKEN_CHARACTER_LITERAL)) then
    return (TRUE);
  elsif (BYPASS(TOKEN_STRING_LITERAL)) then
    return (TRUE);
```

```
  elsif (BYPASS(TOKEN_ALL)) then
    return (TRUE);
  else
    return (FALSE);
  end if;
end SELECTOR;

end PARSER_4;
```

LIST OF REFERENCES

1. Perlis, Alan J., Frederick G., and Shaw, Mary, *Software Metrics: An Analysis and Evaluation* MIT Press, Cambridge, Massachusetts, 1981.

2. Henry, Sallie M., *Information Flow Metrics For The Evaluation Of Operating Systems' Structure* University Microfilms International, Ann Arbor, Michigan, 1979.

3. Parnas, David L., *Information Distribution Aspects of Design Methodology for Computer Systems*. Proceedings of the 1971 IFIP Congress 1971:339-344.

4. Parnas, David L., *A Technique for Software Module Specifications with Examples* Comm. ACM 15, 5:330-336, 1971.

5. Neider, Jeffrey L., Fairbanks, Karl S., *AdaMeasure: AN Ada SOFTWARE METRIC* Naval Postgraduate School, Monterey, California, March 1987.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Dr. Daniel L. Davis Code 52DV Department of Computer Science Naval Postgraduate School Monterey, California 93943	2
4.	Dr. Bruce J. MacClennan Code 52ML Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
5.	Lieutenant Commander Paul Herzig 578-C Schubrick Rd Monterey California 93940	1
6.	Chairman (Code 52) Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
7.	Computer Technology Curricular Officer (Code 37) Naval Postgraduate School Monterey, California 93943	1
8.	Chief of Naval Operations Director, Information Systems (OP 945) Navy Department Washington, D.C. 20350-2000	1

No. Copies

9.	Mr. Carl Hall Software Missile Branch, Code 3922 Naval Weapons Center China Lake, California 93555-6001	1
10.	Mr. Robert Westbrook CMDR 31C Naval Weapons Center China Lake, California 93555-6001	1

END

10-87

DTIC